ANNIS: A graph-based query system for deeply annotated text corpora

Dissertation
zur Erlangung des akademischen Grades

doctor rerum naturalium
(Dr. rer. nat.)

im Fach Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät
der Humboldt-Universität zu Berlin

von
Dipl.-Inf. Thomas Krause

Präsidentin der Humboldt-Universität zu Berlin:
Prof. Dr.-Ing. Dr. Sabine Kunst

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät:
Prof. Dr. Elmar Kulke

Gutachter/innen:
1. Prof. Dr. Ulf Leser
2. Prof. Dr. Anke Lüdeling
3. Prof. Dr. Stefan Evert

Tag der mündlichen Prüfung: 21. September 2018

Acknowledgements

While writing a thesis might sometimes feel like a lonely endeavor, the underlying research is definitely not, and I want to thank everybody that accompanied me on this journey. Specifically, I would like to thank my supervisor Ulf Leser for his continuous support, his numerous precise and helpful comments and his patience. I would also like to express my gratitude to Anke Lüdeling, the head of the Corpus Linguistics and Morphology working group, who made my research possible in the first place and who fostered an environment where work is both challenging and fun. In addition to all former and current colleagues at the Department of German Studies and Linguistics and the other institutions we collaborate with, I would like to especially thank Carolin Odebrecht, Florian Zipser, Heinrich Mellmann and Stephan Druskat for commenting the drafts and giving me valuable feedback. Also, I am grateful for the query logs from the Georgetown University ANNIS server, which Amir Zeldes contributed and which made the benchmarks more representative. I also want to thank all members of my family, friends, and colleagues for enduring me while writing this thesis and for being as great as they are.

The Deutsche Forschungsgemeinschaft (DFG) funded projects LAUDATIO (project number 189321318), and CALLIDUS (project number 316618374) provided funding for the research presented in the following, and I would like to thank for this support as well.

Abstract

The goal of the dissertation is to design and implement an efficient system for linguistic corpus queries. A common task in corpus linguistics is to find occurrences of a certain linguistic phenomenon by analyzing annotations and structures of a so-called annotation graph using a domain-specific query language. The ANNIS Query Language (AQL) is one of these query languages and the ANNIS corpus query system, which is based on the relational database PostgreSQL, implements AQL and has been successfully used for studying various linguistic research questions. ANNIS is focused on providing support for corpora with very different kinds of annotations and uses graphs as unified representations of the different annotations.

For this dissertation, a main memory and solely graph based successor of ANNIS has been developed. Corpora are divided into edge components and different implementations for representation and search of these components are used for different types of subgraphs. AQL operations are interpreted as a set of reachability queries on the different components and each component implementation has optimized functions for this type of queries. This approach allows exploiting the different structures of the different kinds of annotations without losing the common representation as a graph. Additional optimizations, like parallel executions of parts of the query, are also implemented and evaluated. Since AQL has an existing implementation and is already provided as a web-based service for researchers, real-life AQL queries have been recorded and thus can be used as a base for benchmarking the new implementation. More than 4000 queries from 18 corpora (from which most are available under an open-access license) have been compiled into a realistic workload that includes very different types of corpora and queries with a wide range of complexity. The new graph-based implementation was compared against the existing one, which uses a relational database. It executes the workload ~ 10 faster than the baseline and experiments show that the different graph storage implementations had a major effect in this improvement.

Zusammenfassung

Das Ziel dieser Dissertation ist es, ein effizientes System für die Suche in linguistischen Korpora zu designen und implementieren. Eine häufige Aufgabe in der Korpuslinguistik ist es, Vorkommen von bestimmten linguistischen Phänomenen zu finden, indem die Annotationen und Strukturen eines sogenannten Annotationsgraphen mit Hilfe einer domainenspezifischen Anfragesprache durchsucht werden. Die ANNIS Query Language (AQL) ist eine dieser Anfragesprachen und das ANNIS Korpussuchsystem, das auf der relationalen Datenbank PostgreSQL basiert, implementiert AQL und wurde bereits erfolgreich für verschiedene linguistische Studien verwendet. ANNIS is spezialisiert darin, Korpora mit verschiedenen Arten von Annotationen zu unterstützen und nutzt Graphen als einheitliche Repräsentation der verschiedener Annotationen.

Für diese Dissertation wurde eine Hauptspeicher-Datenbank, die rein auf Graphen basiert, als Nachfolger für ANNIS entwickelt. Die Korpora werden in Kantenkomponenten partitioniert und für verschiedene Typen von Subgraphen werden unterschiedliche Implementationen zur Darstellung und Suche in diesen Komponenten genutzt. AQL-Operationen werden als Kombination von Erreichbarkeitsanfragen auf diesen verschiedenen Komponenten implementiert und jede Implementierung hat optimierte Funktionen für diese Art von Anfragen. Dieser Ansatz nutzt die verschiedenen Strukturen der unterschiedlichen Annotationsarten aus, ohne die einheitliche Darstellung als Graph zu verlieren. Zusätzliche Optimierungen, wie die parallele Ausführung von Teilen der Anfragen, wurden ebenfalls implementiert und evaluiert. Da AQL eine bestehende Implementierung besitzt und diese für Forscher offen als webbasierter Service zu Verfügung steht, konnten echte AQL-Anfragen aufgenommen werden. Diese dienten als Grundlage für einen Benchmark der neuen Implementierung. Mehr als 4000 Anfragen über 18 Korpora (von denen die meisten unter einer Open-Access Lizenz zur Verfügung stehen) wurden zu einem realistischen Workload zusammengetragen, der sehr unterschiedliche Arten von Korpora und Anfragen mit einem breitem Spektrum von Komplexität enthält. Die neue graphbasierte Implementierung wurde mit der existierenden, die eine relationale Datenbank nutzt, verglichen. Sie führt den Anfragen im Workload im Vergleich ~ 10 schneller aus und die Experimente zeigen auch, dass die verschiedenen Implementierungen für die Kantenkomponenten daran einen großen Anteil haben.

Contents

1. Introduction

This work describes a query system for annotated text corpora, which in this context are texts enriched with various types of annotation and collected for linguistic research. Such a corpus linguistic methodology is nothing new, for example in Meyer (2002) it is stated that:

> "If a corpus is defined as any collection of texts (or partial texts) used for purposes of general linguistic analysis, then corpus linguistics has been with us for some time." (Meyer 2002, p. xii)

Concordances (lists of all or a selection of principal words of a text in their context) of the Bible have already been created per hand in the 18th century, and also earlier dictionaries are sometimes based on systematic collections of text (Meyer 2008). Early projects to create searchable concordances with the help of computers date back to the very early beginning of computing. The first one was a project of Roberto Busa (supported by IBM) to create a digital concordance of the works of Saint Thomas Aquinas which started in 1946 (Wynne 2008) and still can be searched and viewed online[1]. Compared to manually created corpora, modern digital corpora allow a much more flexible representation of their content and on-demand automatic extraction of structured information using the appropriate software.

This chapter introduces some basic corpus linguistic annotation concepts, shows how query systems can be used to support linguistic research and motivates the development of a new query engine named graphANNIS, which replaces a relational database in the existing query system ANNIS (Krause and Zeldes 2016). Later chapters will give an overview of the state of the art in the field of corpus query systems (Chapter 2) and describe existing data models for linguistic annotations, including a domain specific corpus language (Chapter 3). Based on this previous work, a novel, graph-based and search-optimized data model for linguistic corpora is presented (Chapter 4), the design and implementation of a query system based on this data model (Chapter 5) and several optimization techniques (Chapter 6) are described in more detail. The implementation and optimization techniques are evaluated in Chapter 7, followed by a conclusion and outlook in Chapter 8.

1.1. Corpus linguistics and annotations

Corpus linguistic methods can be used for analyzing the structure of language and how language is used (Biber et al. 1998, p. 1). It is based on collections of expressions of speech, which can have various forms. Speech can be collected on different mediums:

[1] `http://www.corpusthomisticum.org/it/` (last accessed 2017-12-14)

It can be a native digital text (written for example in a word-processor or as part of a web-page) but also a traditionally printed or handwritten text. Speech can not only be represented as text. For example, spoken language can be persisted as audio or video recordings. For the study of sign language, even multiple synchronized video recordings are used (Hanke et al. 2010). The non-digital or non-written expressions of speech have to be digitalized and transcribed before they can be used in a digital corpus. Another aspect of corpora is the speaker itself. For example, there can be a single speaker or several ones which are either in some form of dialog or are the collective author of a coherent text. These speakers can also be described by metadata (for example age and language acquisition history) which together with metadata for the text itself provides more context to the situation of the speech production and are important for understanding and interpreting the observed linguistic phenomena. The given examples of possible sources for digital corpora are far from complete, and there are also much more aspects of the expression of speech that can be important for a specific linguistic research question.

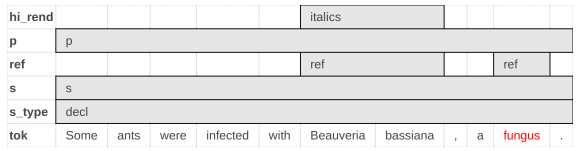
What the different kind of linguistic corpora have in common is, that they can allow a systematic and empirical evaluation of linguistic phenomena. The digital texts are enriched with annotations for this purpose. These annotations can mark linguistic phenomena, properties of the text itself, its author(s) and other information relevant to the research question. Annotations can take different forms by themselves. For example, they can be attached to the words of the text, to ranges of text or point of times, they can form hierarchies, and there can also be non-hierarchic references. A small set of examples for linguistic annotations are shown in Figure 1.1. These annotations are typically rooted in the framework of a linguistic theory, and the examples in Figure 1.1 show visualizations of these annotations. Such visualizations can be used by linguists to describe how a given linguistic theory is applied to an example text. In corpus linguistics, however, these visualizations are representations of more general descriptions of structured annotations. Different linguistic theories can use the same underlying structural principles but use different category names and values. The empirical evaluation of linguistic phenomena is typically based on finding instances of expressions of speech that match a certain common pattern on these annotations and by comparing the results qualitatively and/or quantitatively.

While corpus linguistics itself is used as a method by linguists to investigate genuine linguistic research questions, there is an overlap with the research field of computational linguistics or Natural Language Processing (NLP) (Dipper 2008). Corpora can be used as training material for machine learning techniques, and the resulting NLP tools are often used to annotate corpora automatically. However, a huge amount of annotation is done manually by linguists. Because creating corpora manually is such a large amount of work, they are typically non-volatile data. When using a query system, a specific released version of a corpus is imported once into the system and then used for read-only queries.[2]

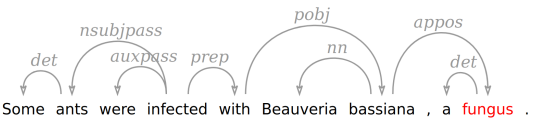
[2]For corpora that are entirely automatically annotated, the workflow can be more flexible, for example with scripts that add custom annotations based on previous ones before doing the actual analysis.

Some	ants	were	infected	with	Beauveria	bassiana	,	a	fungus	.
DT0	NN2	VBD	VVN	PRP	NP0	NN1	PUN	AT0	NN1	PUN
some	ant	be	infect	with	Beauveria	bassiana	,	a	fungus	.
DT	NNS	VBD	VBN	IN	FW	FW	,	DT	NN	.
DT	NNS	VBD	VVN	IN	FW	FW	,	DT	NN	SENT
det	nsubjpass	auxpass	root	prep	nn	pobj	punct	det	appos	punct

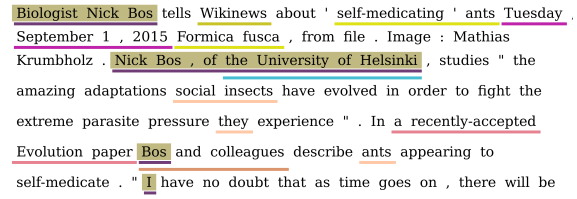
(a) Annotations attached to each token (words or punctuation), displayed as Keyword in Context (KWIC)



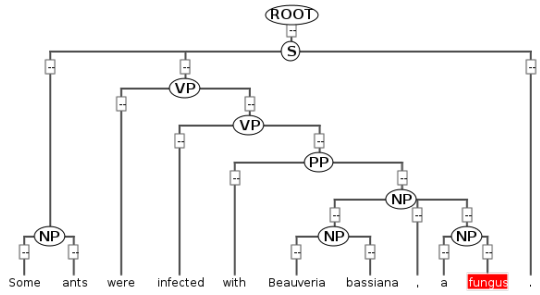
(b) Annotations for a range of tokens (spans).



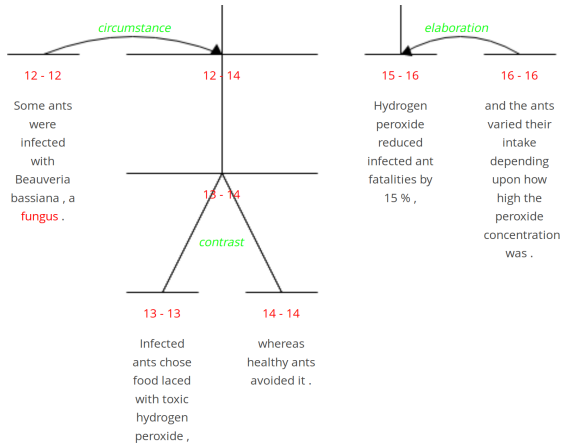
(c) Tokens connected with edges, where the edge label corresponds to the syntactic feature of the tokens in the sentence.



(d) Edges between spans (marked by the same color of the line) inside a complete text, marking references to re-occurring entities.



(e) Syntactical structure of a sentence expressed as hierarchy.



(f) Hierarchical structure of sentences.

Figure 1.1.: Examples for linguistics annotations from the GUM corpus (Zeldes 2016b). A simple form of annotation is to assign key-value pairs to each token (a), for example for marking the part of speech of a word or its lemma. Similarly, annotations can be assigned to ranges of tokens called spans (b). In (c), the syntactic structure of a sentence is represented by connecting tokens with an edge and using the syntactic function as a label on the edges (this is called a dependency tree because the edges construct a hierarchy of the tokens). So-called co-referents (which mark re-occurring entities), can be modeled with edges between spans (d). Hierarchies can be constructed for the elements of a sentence (for example, the constituent tree (e) shows the syntactic structure of a single example sentence), but such hierarchies can also be constructed with sentences as leafs (for example in the Rhetorical Structure Theory (RST) example (f)).

1.2. Using corpus queries for empirical research

This section aims to provide a more practical insight into how corpora can support linguistic studies and how the process of a typical corpus study looks like. It is based on an actual corpus linguistic study presented in Goschler (2014) as an example. The study deals with the linguistic phenomena of agreement of subject and verb in German and its variation, which is explained in more detail below.

Agreement means that the verb and the subject both have the same grammatical person and number. For example, in the sentence

Example 1.1 **One person* have traveled to Berlin.

the grammatical number of the subject "One person" and verb "have" does not agree and the whole sentence appears to be ungrammatical (which is marked by the star). In German, agreement is assumed to be highly related to the syntax. The study in Goschler (2014) examines the agreement for conjuncts (phrases of nouns that are connected with a conjunction like "and") in German in more detail. It starts with the observation that there are cases where the verb is in singular even if its subject is a conjunct of two singular nouns. The expectation would be that the conjunct would trigger a plural verb. For example, in

Example 1.2
(a) *Berlin und Spandau* sind umgeben von Brandenburg
(Berlin and Spandau are surrounded by Brandenburg)
(b) **Berlin und Spandau* ist umgeben von Brandenburg
(*Berlin and Spandau is surrounded by Brandenburg)

the conjunct of two entities together is a plural noun phrase, and thus only the form where the verb is plural is grammatically correct (a) and the singular form is ungrammatical (b). In contrast, the example

Example 1.3 (from the TIGER corpus (Brants et al. 2004))
Regieren und Herrschen ist strafrechtlich nicht als bloßes Unterlassen zu würdigen
(Governing and ruling are not to be regarded under criminal law as mere omission.)

shows that it is possible to have a singular verb even for conjuncts. To test if this form is rare and to find out the contexts in which this form occurs, a corpus study can be conducted. Such a study can provide more examples for manual examination and a statistical analysis. In Goschler (2014), a concordance software was used to retrieve all instances of "und" with their context in a corpus, and then each of the 35.383 instances was manually checked

- if both nouns form a complete phrase (there are no more other words in the phrase),
- that the nouns are actually in singular form, and
- that the noun phrase is the subject of the sentence.

Wir	wollen	mehr	Arbeitsplätze	und	mehr	Zeitsouveränität	
PPER	VMFIN	PIAT	NN	KON	PIAT	NN	
durch	Freizeitausgleich	für	Mehrarbeit	und	Arbeitszeitkonten	.	
APPR	NN	APPR	NN	KON	NN	$.	

(We want more jobs and more time sovereignty through leisure-time compensation for overtime and working time accounts.)

Figure 1.2.: Example part of speech annotation for a conjunct taken from the TIGER corpus (Brants et al. 2004). Each token has assigned an annotation value for its part of speech, which is encoded by an abbreviated element of the tag-set. For example, nouns are labeled with "'NN", and conjunctions are labeled with "KON".

Each instance was then categorized if the verb was in singular or plural.

If the corpus used in the study would not only consist of digitalized text but would also be enriched with relevant linguistic annotations, this search could be done more efficiently. Part of speech information for each word of the corpus could be used to filter the matches to only include instances where a noun is both on the left and right side of the conjunction "und". Preprocessing of the corpus would need to include

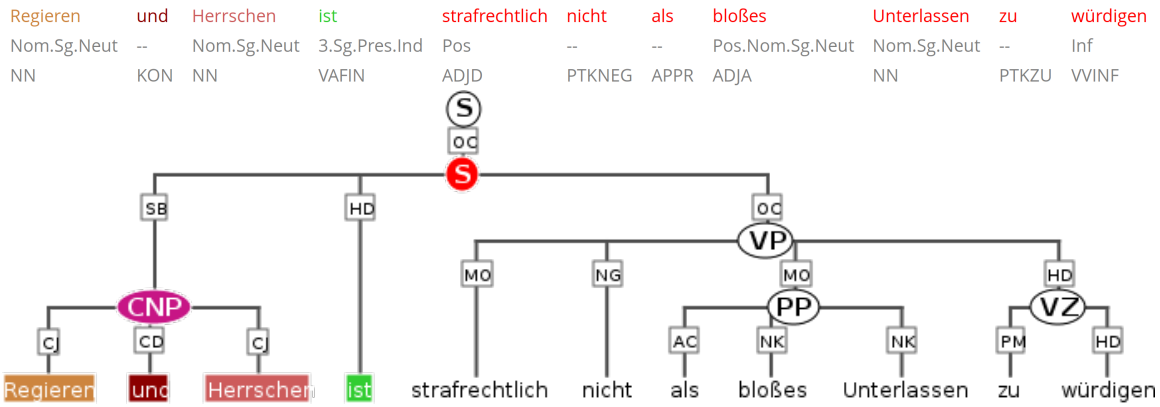
- dividing the text into tokens (one token for each word or punctuation),
- adding an explicit ordering for the tokens to establish a concept of "comes before,", and
- assignment of part of speech annotation to each token with a specified set of labels (a so-called tag-set).

An example match that would be returned by filtering the corpus for sequences of a noun, the conjunction "und" ("and") and another noun can be seen in Figure 1.2. This example includes two conjuncts, but only one ("Mehrarbeit und Arbeitszeitkonten") is directly preceded and followed by a noun. The other one ("mehr Arbeitsplätze und mehr Zeitsouveränität") has more complex noun phrases which include the pronoun "mehr" ("more"). Also, not all the nouns are actual in singular, as it is requested in one of the conditions. If this were the results of a search query against a corpus, either additional manual filtering or a refinement of the search query would be required. To solve this problem, morphological annotations, like in the exemplary Figure 1.3, can be used as additional filter. In this particular case, the annotation scheme uses the same label for different kinds of morphological information. Since the other information is not relevant to the phenomenon, regular expressions are needed to filter out instances where the label matches a certain pattern. For example, the regular expression `.*\.Sg\..*` would find all words in singular and the regular expressions `.*\.Pl\..*` all words in plural form. Another problem present in the example in Figure 1.2 is that one of the conjuncts is not actually the subject of the sentence. This can be solved with linguistic annotations of the syntax of the sentence, like in Figure 1.4 where different parts of the sentence are put in relation to each other with

Dagegen	hatten	Verteidigung	und	Staatsanwaltschaft	Revision	eingelegt	.
--	3.Pl.Past.Ind	Nom.Sg.Fem	--	Nom.Sg.Fem	Acc.Sg.Fem	Psp	--
PROAV	VAFIN	NN	KON	NN	NN	VVPP	$.

(Defense and the public prosecutor's office had filed an appeal against this.)

Figure 1.3.: Example morphology annotation for a conjunct taken from the TIGER corpus (Brants et al. 2004). This KWIC includes also part of speech annotation but has an additional layer of morphology information. The morphology label consists of multiple parts separated by dots, where each part describes a different morphological aspect and the second part describes if a word is singular ("Sg") or plural ("Pl").



(Governing and ruling is not to be regarded under criminal law as mere omission.)

Figure 1.4.: Example syntax annotation for a conjunct taken from the TIGER corpus (Brants et al. 2004). The syntax annotations form a hierarchy of the different parts of the sentence. Each part has a label with its category, for example, "S" for sentences and "CNP" for conjunct nominal phrases. Labels at the edges describe an additional function of the different parts of the sentence. For example, the edge label "SB" marks the subject of the sentence and "HD" the head of a phrase. For this complete sentence, the head is the finite verb. Part of speech and morphological annotations for each token are shown as well because they are also needed to find instances of the example phenomenon.

.

edges and additional hierarchical nodes. The query can now be filtered only to include patterns where the nouns and the conjunction belong to the same phrase, and where this phrase is the subject of the sentence. Morphology annotation can be used again to determine if the verb for the subject is in plural or singular form. In the study of Goschler (2014), it could be shown, that there is a high variation in using either the plural or singular verb form for these kinds of constructions and that the latter one is not rare at all. It was also shown, that this phenomenon is not new, but has been present in earlier stages of the German language.

This example demonstrates that it is possible to combine different existent linguistic annotations in a query to filter for more specific linguistic phenomena. By combining different annotations into a single corpus and by providing a unified interface to analyze them, a so-called *multi-layer* or *multi-level corpus* (Dipper 2005) is created. Such corpora are very useful for various field of linguistic studies, including but not limited to the study of learner language (Lüdeling et al. 2005), spoken dialogues (Rodríguez et al. 2007) or historical texts (Odebrecht et al. 2017). While it is cumbersome to generate these annotations either manually or with automatic tools, they can be used flexibly and help to answer a wide range of research questions, even ones that have not been considered when the corpus was created initially.

A research process like this is typically iterative, a user begins with simple queries and refines them to be more inclusive or specific for a phenomenon. Thus, it is important that the query execution is as responsive as possible: the user can not wait a long time for a query to execute and then find out that the result did not match his expectations and needed additional refinement. Also, the result of the queries can be used in different ways (users can perform additional manual filtering and evaluation, statistical tests or even more automatic post-processing) and thus the visualization of the annotations and their export must be possible.

1.3. Common problems of linguistic query systems

Query systems for linguistic corpora face several challenges. They have to support different types of corpora and must be generic enough to allow a wide variety of linguistic phenomena to be articulated in their query language. But they also have to make sure that a user is able to actually handle the complexity of the system and its query language. That is why corpus query systems often include their own query languages (a selection of them is discussed in Section 2.3) in the form of a Domain Specific Language (DSL) or use very specific graphical interfaces. These DSLs typically use terms and concepts familiar to linguists, but standardization or re-usage of existing query languages is just in its infancy. Also, more specific query systems and query languages often only support a subset of relevant types of annotation. Since it is useful to combine different types of annotation in one query to describe more and more complicated phenomena (such as the one in Section 1.2), such restrictions can become a problem for the researcher. Still, specialization to certain types of annotations or even to a single corpus allows providing faster query systems, for example by partitioning corpora into small documents or even into sentences. Generalizing over all these annotations and corpora can be done with graphs and is described in more detail in

Chapter 3 and 4. An existing solution which is based on such annotation graphs is the ANNIS system with its query language "ANNIS Query Language" (AQL) (Rosenfeld 2010; Krause and Zeldes 2016). ANNIS uses a relational database to execute the queries and while it has been used in production over several years and by numerous researchers[3], the approach of mapping annotation graphs to a relational data model has been shown to be problematic for performance and scalability reasons (see Section 3.3).

These problems lead to the question of how a generic multi-layer query system, supporting most types of linguistic annotations, can be implemented efficiently, with better execution times than the current generic query systems. In this dissertation, a novel graph-based approach is proposed, which partitions the corpora into edge components instead of documents and which uses specialized implementations for storing different types of graphs in a common main memory query system. To distinguish this novel implementation from the existing approach, the new query engine is called graphANNIS, while the existing relational database implementation is referred as relANNIS. The overall system, which also includes a web-based user interface and the query language AQL, is referred to as ANNIS.

[3]It is difficult to measure the total number of users since ANNIS is open-source and everyone is free to install and use it. There is also no central ping-back service which would count the number of installations or users for privacy reasons. Currently, there are at least 14 public installations of ANNIS that the author is aware of (see Section A.2 in the appendix for a list). In addition, users can choose to host private installations on servers or their own desktop computer. The public available ANNIS server from the Corpus linguistics and morphology research group at the Humboldt-Universität zu Berlin (`https://korpling.org/annis3`, last accessed 2017-10-25) hosts more than 150 corpora, has 96 registered users and serves roughly 1800 queries every month.

2. Background and related work

This chapter gives an overview of existing work which is relevant when implementing a multi-layer linguistic query system. First, it motivates why graphs can and should be used to encode linguistic information Then, different technologies that are used to query graphs, or linguistic corpora are presented. This can not be a complete description of all technologies in the field of corpus linguistics since such a list would be extremely long, given the amount of research that is done in this field. Nevertheless, it hopefully still shows which main implementation strategies have been in use and proven to be useful.

2.1. Representing linguistic annotations as graphs

Linguistic theories and models of language can be described in a large variety of formal models. By using computers to process the linguistic information it becomes necessary to encode this information in some machine-readable way. As long as speech acts are only represented as text and no further structural information needs to be encoded, it is sufficient to represent text as a stream of characters in a simple text file. But structural information is needed to encode even most basic linguistic concepts like words, and thus these concepts have to be encoded in file formats readable by the tools that are used to process these files. Computer-aided linguistic research has led to a diverse landscape of tools and formats.[1] Encoding the segmentation into words or tokens can be achieved, for example, by writing each token into one line. The popular part-of-speech tagger TreeTagger (Schmid 1995) uses this format to encode the results of its tokenization preprocessing step. By using tab-separated columns, the TreeTagger than adds additional annotations to each token. An example is given in Figure 2.1. This simple format can be extended to allow additional SGML-tags around the lines to add annotations to a continuous set of tokens. Another popular format is the Penn Treebank Bracket (PTB) format (Bies et al. 1995). It encodes the tokens and a hierarchical tree structure by using brackets to enclose the children of a parent node (an example is given in Figure 2.2). Other formats like EXMARaLDA (Schmidt and Wörner 2014), TigerXML (Mengel and Lezius 2000) or PAULA (Dipper 2005), are often based on XML and can cover a wide range of different annotation structures, including but not limited to

- time-based alignment with audio and video files,
- syntax trees,

[1] The huge number of different annotation file formats has been once coined as "multiverse of formats" (Zipser et al. 2011) or "Formatpluriversum" (Zipser 2014) by a developer of a conversion tool for such formats.

```
That      DT      that
is        VBZ     be
a         DT      a
Category  NN      category
3         CD      3
storm     NN      storm
.         SENT    .
```

Figure 2.1.: Example for the tab-based TreeTagger format. Each line represents a token and its annotations (in this case part-of-speech and lemma).

- references to tokens inside a text, and
- multiple layers of different annotations.

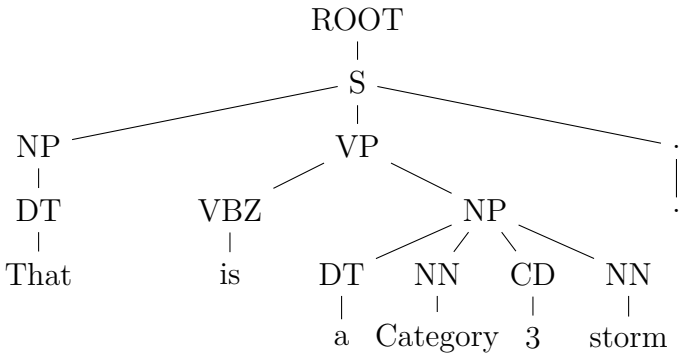
The formats used by the tools are all based on different linguistic theories but are not necessarily based on an explicitly described data model. This makes it difficult to compare and convert formats into each other because the formats do not necessarily follow a common terminology or are based on the same structural components. In Bird and Liberman (2001), a more abstract and general framework for linguistic annotations is described. It is based on directed graphs and influenced the development of other models and data formats. Especially, it was influential for the ISO standard "Linguistic Annotation Framework" (LAF) (ISO:24612 2012), which is based on graphs, too. LAF is intended a "basis for harmonizing existing language resources as well as developing new ones" (Ide and Romary 2004). A data format that implements the LAF model is GrAF (Ide and Suderman 2007). The ISO standard is an important indication that graphs can be used to represent the linguistic annotations that are used today, but "they do not address the mapping between themselves and the already used formats" (Zipser and Romary 2010). Mapping existing linguistic annotations can be achieved with the conversion tool Pepper and the meta-model Salt (Zipser and Romary 2010). The idea of Pepper is to map all import and output formats onto the same intermediate meta-model and to provide a programming API to create instances of this meta-model. It is not intended as persistent data format, and its ephemeral character allows to extend it more easily without breaking backward-compatibility. Salt can also map GrAF and is therefore compatible to LAF. The original relational database implementation of ANNIS and Salt have been developed in parallel and Salt is used as interchange model for the ANNIS web-services. That is why it is also used as a base for the novel graphANNIS data model. Section 3.1 describes how Salt maps different linguistic annotations in more detail and Chapter 4 describes the graph-based data model of graphANNIS.

2.2. Graph query languages and implementations

This section will give an overview of some existing graph query languages and their differences and similarities. It is far from complete and concentrates on graph query implementations that have been proposed or used for linguistic query systems.

```
(ROOT
        (S
        (NP (DT That))
        (VP (VBZ is)
        (NP (DT a) (NN Category) (CD 3) (NN storm)))
        (. .))
)
```

(a)



(b)

Figure 2.2.: Example for representing a hierarchical annotation with the Penn Treebank Bracket (PTB) format. In (a) the content of an example PTB file is given and (b) shows its visual representation.

In Wood (2012) a survey of general application graph query languages is given. While some languages like PQL (Leser 2005) or BiQL (Dries et al. 2009) are influenced by relational algebra and "Structured Query Language" (SQL) syntax, others like GraphLog (Consens and Mendelzon 1990) originate from logic programming. Regular paths, as they have been described in Mendelzon and Wood (1995), are a frequently used concept in the described languages. Given a path between two nodes of a graph, each edge of the path can be labeled with a symbol from an alphabet. Speaking in corpus linguistics terms, the alphabet consists of the possible values for a specific annotation category and each label of an edge corresponds to an annotation. A regular path query defines a regular language on this alphabet, and a path is included in the result if the concatenation of its labels is part of the regular language. For query languages that are based on relational algebra, such regular paths or similar concepts are an important extension for describing the results of transitive closures, something that is not possible with the traditional definition of the relational algebra (Aho and Ullman 1979).

2.2.1. RDF based databases and SPARQL

Representing graphs in databases is relevant for different fields, but with the introduction of the highly interconnected World Wide Web, representing the relations between resources, storing them in databases and querying these relations have fostered the creation of the Resource Description Framework (RDF) (Carroll and Klyne 2004).

RDF is a graph-based data model which focuses on the relations between the resources. Thus, graphs are a set of triples, where each triple represents an edge in the graph. The triple has a subject (the source node), an object (the target node or value) and a predicate, which is a more detailed description of the relation. While different data types are allowed as object values, only resources (represented via Uniform Resource Identifiers (URIs)) are allowed as subjects. Since a graph is purely defined by a single type of data structure (the triples), it is easy to define subsets of a graph. It would be possible to map RDF graphs to labeled directed graphs with their different kind of elements, for example by having triples for each label of a node, with a special "is label of node" predicate.

If the graph is modeled in RDF, an implementation of the standardized "SPARQL Protocol and RDF Query Language" (SPARQL) (Harris and Seaborne 2013) can be used to search the graph. Regular path queries were introduced in version 1.1 of SPARQL under the term "property path queries." The standardization process of property path queries in SPARQL made it clear that it is necessary to carefully select the features which are supported by the query language, to allow an efficient query processing implementation. In Arenas et al. (2012) and Losemann and W. Martens (2012) it was shown that the original semantics of counting each path between two nodes as a separate result would prohibit any efficient implementation. As a result of these findings, the semantics were changed to check for the existence of any path instead. Also, the possibility to specify the number of repetitions was removed.[2] Implementing the regular path queries was challenging, for example in Gubichev et al. (2011) an implementation based on Dijkstra's algorithm was implemented on top of the triple store RDF-3X (T. Neumann and Weikum 2010) and compared to the implementation of the RDF reference implementation Jena (Carroll et al. 2004). Three manually selected queries on a dataset with 845 million triples where executed using Dijkstra's algorithm with an average response time of 2.88 seconds, while the Jena implementation was stopped at 30 minutes.

RDF has been used to represent linguistically annotated corpora. For example, the POWLA format (Chiarcos 2012) is a realization of the principle data model of the PAULA format into RDF, and there exists a tool to map existing corpora encoded in PAULA to POWLA. Chiarcos et al. (2013) propose to use RDF for linking corpus resources to existing resources of different types, like lexicons or metadata entries. Since SPARQL is generic and not restricted to linguistic corpus search, it can be used to analyze these different types in the same query. However, like with XML based formats, conflicting representations of corpus data in RDF can exist and the queries can be only written with the knowledge of the underlying specifications. While it could be possible to map queries from domain-specific linguistic query languages like AQL to RDF, this mapping would only work for corpora encoded with the same scheme. Mapping a labeled directed graph to RDF will lead to an increase in triples and more joins in the queries because patterns on labels for nodes and edges need to be expressed as conditions on triples that are connected with the node or edge subjects. Also, the general purpose SPARQL query systems have no knowledge of

[2]The discussion which led to the semantic change can be followed here: `http://lists.w3.org/Archives/Public/public-rdf-dawg-comments/2012Apr/0003.html` (last accessed 2017-10-25)

domain-specific properties of the transformed annotation graphs and thus can not use this knowledge for optimizations.

2.2.2. Neo4j and the Cypher query language

Neo4j (Robinson et al. 2013) is a graph database which is based on the property graph (Rodriguez and Neubauer 2010) model and uses the query language Cypher. Cypher has been further developed as a general query language not tied to Neo4j and with an open specification under the name openCypher (Marton et al. 2017). It allows specifying graph patterns with variable path lengths, which makes it comparable to regular path queries.

In Robinson et al. (2013), it is described in detail how the nodes, edges, and labels are represented on physical storage. Neo4j uses separate stores for nodes and relations. The size of each entry is fixed to 9 bytes for each node and 33 bytes for each relationship. Having fixed tuple sizes allows accessing the properties of a certain tuple without any lookup index in constant time. The authors of Neo4j claim that their data structures are optimized for graph traversals (Robinson et al. 2013, pp. 144 ff.). In order to achieve this, the relation tuples not only contain the ID of the nodes they are connected to, but also the IDs of the connected relations. Having a fast mechanism to traverse the graph is obviously an advantage for certain types of queries, but only relying on traversal as single implementation strategy might be a disadvantage for others. An example could be queries that have both very infrequent labels on the start and end node but are connected by numerous paths. A traversal could make use of the small number of start nodes, but could not exploit the small number of end nodes.

Neo4j has been proposed to as back-end for linguistic query system. For example, the National Corpus of Polish has been integrated into Neo4j (Pęzik 2013). Also, it was proposed to add a back-end based on Neo4j to the corpus query system KorAP (Bański et al. 2013). However, despite the similarity of the data models to linguistic annotation graphs, to the best of our knowledge, there is no generalized open-source corpus system based on Neo4j published yet.

2.2.3. Approaches to using relational databases as graph database

Relational database management systems (RDBMSs) are used in a variety of domains to store and query data, and numerous commercial and academic systems are available. Because of the general applicability and maturity of these systems, they are also used to represent graphs. Examples for using RDBMSs as graph database are implementations like AgensGraph[3], which is based on PostgreSQL[4] and allows openCypher queries, or Sqlg[5] which acts as back-end for the Gremlin graph traversal language (Rodriguez 2015) and can be used with a various number of RDBMSs. RDF storage and query

[3] `http://www.agensgraph.com/` (last accessed 2017-10-25)
[4] `https://www.postgresql.org/` (last accessed 2018-02-17)
[5] `http://www.sqlg.org/` (last accessed 2017-10-25)

systems also use RDBMSs as back-ends, for example, the Jena system (Wilkinson et al. 2003) or Virtuoso (Erling 2012), which implements RDF on top of a column-store. These RDBMS based systems can use the relational database as a pure storage back-end and have some kind of graph-aware API on top of it, or they can also use SQL to allow regular path queries. Different strategies have been proposed to circumvent the shortcoming of the original relational algebra, which does not allow querying the transitive closure of an operator and thus can not directly implement queries for paths of unknown length. Recursive SQL can be used when the graphs are encoded as adjacency list and Ordonez (2010) presents optimization strategies to speed up recursive SQL queries. Another approach is not to use adjacency lists for reachability queries, but special graph indexes which encode reachability information as property of relational tuples. Instead of using recursive joins, they express reachability by directly comparing the index values of two tuples. Examples of these kinds of graph indexes are pre- and post-order encoding (Grust et al. 2004), GRIPP (Trissl 2012), Grail (Yildirim et al. 2010), and Ferrari (Seufert et al. 2013). Section 3.3.1 will explain pre-/post-order encoding and its applicability for encoding reachability in relational databases in more detail.

2.3. Linguistic query languages and implementations

Given the various already existing well-defined general graph query languages, the question arises if it is necessary to develop and maintain a Domain Specific Language (DSL) like the ANNIS Query Language (AQL), which shares similarities with more general graph-based languages. In Deursen et al. (2000) several typical benefits and disadvantages of DSLs are compiled. Such a benefit is that it is easier for the user to formulate their questions in a language that uses the idioms of their research field like "annotations" instead of labels. Given the high complexity of general graph queries, it is also desirable to restrict queries to search operations that can be efficiently implemented. The disadvantage is that there is no established general linguistic query language yet and users need to learn a new query language for each new query system. A current initiative wants to change this situation, by developing a "lingua franca" of linguistic query languages and propose it as an ISO standard (Frick et al. 2012). Following, existing corpus query systems and query languages are presented.

2.3.1. Representing large token-centered corpora with the IMS Open Corpus Workbench

The IMS Open Corpus Workbench (CWB) (Evert and Hardie 2011) is a system for corpus analysis and its main component, the Corpus Query Processor (CQP), is a popular concordance system. Concordances are described in more detail in Wynne (2008):

> "A concordance is a listing of each occurrence of a word (or pattern) in a text or corpus, presented with the words surrounding it. A simple concordance of 'Key Word In Context' (KWIC) is what is usually referred

> to when people talk about concordances in corpus linguistics […]" (Wynne 2008, p. 710)

CWB is implemented in the C programming language and has been developed since the early 1990s (Christ 1994). It has been used as back-end for web applications like CQPweb (Hardie 2012), and numerous corpora are available for CWB. Its data model is optimized for representing token annotation (so-called position based annotations), but CWB also allows encoding and querying of spans (annotations that cover a range of tokens) which are called structural annotations by the authors of CWB. Annotation values and word-forms can be searched with the help of regular expressions, but the query language of CQP also allows expressing regular patterns on tokens. For example, it is possible to search for a non-specified number of tokens between two other tokens. The position of an annotation in the "stream of sequential token positions" (Evert and Hardie 2011, p. 6) of a corpus is the central anchor point for all annotations and indexes. Queries are evaluated by converting the regular patterns on tokens to a finite state machine. A search always starts by finding the first token of the pattern and then uses the finite state machine to match the complete query. This allows to utilize the locality of tokens to each other, but it leads to problems when the definition of the first token of the query matches too many instances.

2.3.2. TIGERSearch and other treebank search tools

Treebanks (Nivre 2008) are corpora that, in contrast to the "flat" spans which can be represented in CWB, contain explicit hierarchical structures as a result of some kind of (often manual) grammatical analysis. These hierarchical structures are not necessarily always trees in a graph-theoretical sense but can be described as Directed Acyclic Graphs (DAGs) (Cormen et al. 2009, p. 1172). Example corpora are the Penn Treebank (Marcus et al. 1993), which used the bracketing format described in Section 2.1, the TIGER corpus (Brants et al. 2004), which has crossing edges and different edge types, and TüBa-D/Z (Telljohann et al. 2009).

The Penn Treebank included a tool called tgrep which was later re-implemented as TGrep2 (Rohde 2005). TGrep2/tgrep uses the terms "dominate" for a parent-child relationship between two annotation nodes. A node can "immediately dominate" another node if it is the parent node and "dominates" it if the other node is a descendant. TGrep2 is implemented in the C programming language. It operates by translating the bracket format into a binary representation of the trees once and writing this representation to disk. Queries are executed by iterating over all trees separately, and only one tree (in addition to its limited context) is loaded into memory at the same time. A tree is returned if it matches the pattern, similar to the `grep` command under Unix/Linux. Since each tree is limited to one sentence, the search space for reachability queries is small.

TIGERSearch (Lezius 2002) is also based on a treebank search tool, which utilizes the limitation of the intra-sentence-only annotations. It was developed as main search tool for the TIGER corpus and is implemented in Java. In contrast to TGrep2, it allows crossing edges and a second edge type. This cannot be represented in the bracketing format PTB but in the TIGERSearch base format TIGER-XML. Corpora

used by TIGERSearch need to be encoded in TIGER-XML and will be indexed once. When executing a query, this index is used to filter for sentences that have the needed annotation values somewhere in the annotation graph. For these sentences, the query is evaluated on the complete graph similar to TGrep2. While TGrep2 is a command line tool, TIGERSearch comes with a complete graphical user interface. It is not only used to visualize the resulting trees, but it also contains a graphical query builder for its idiosyncratic query language. The TIGERSearch query language has been re-used in the web-based TüNDRA system (S. Martens 2012), which translates TIGERSearch queries into XQuery and executes them using the BaseX XML database (Grün 2006).

While TIGERSearch is based on a custom graph matching engine, the VIQTORYA query tool (Steiner and Kallmeyer 2002) is based on the MySQL[6] relational database. Trees are mapped to a relational database schema, and the query language of VIQTORYA is translated to SQL. Edges are represented by an adjacency list, and there is no operator for querying reachable nodes with an arbitrary path length: Only direct dependents can be queried. VIQTORYA also features a graphical user interface for constructing queries.

2.3.3. Querying multi-layer corpora with ANNIS

ANNIS (Krause and Zeldes 2016) is a query system that aims to support not only one type of annotation (like trees or spans), but integrates the different types of annotations into one query language and search tool. Such a corpus with multiple annotation layers, which contain different linguistic analysis, is called a multi-layer or multi-level corpus (Dipper 2005). It is based on the PostgreSQL[7] relational database in its original implementation (Rosenfeld 2010). The query language is inspired by the one of TIGERSearch, and like TIGERSearch it also includes an operator for indirect dominance, which means it must implement queries for reachable nodes efficiently. This is achieved by using pre- and post-order encoding, as proposed by Grust et al. (2004). A problem of the relational database implementation of ANNIS is that it is not optimized for larger corpora. While a treebank with the size of the TIGER corpus is still feasible, larger ones like TüBa-D/Z are already problematic. Being a multi-layer corpus search tool, the database size is not only growing when new documents with new tokens are added, but also because corpora integrate more and more annotation layers. The way ANNIS stores the data in PostgreSQL does not scale well with this type of additions (a more detailed description of the relational database based implementation of ANNIS is given in Section 3.3). An approach for better scaling of the relational model was described and implemented in Rosenfeld (2012), by using the column-oriented data-store MonetDB instead of PostgreSQL. This approach still relies on pre- and post-order as the only way of encoding reachability.

ANNIS is not only a query tool, but it also aims to provide visualizations for the very different types of annotations it supports. For this, a browser-based user interface has been developed (Hütter 2008), which can be extended with plug-ins for new types of visualizations. The user interface is connected to a REST-based server application.

[6] `https://www.mysql.com/` (last accessed 2017-10-25)
[7] `https://www.postgresql.org/` (last accessed 2018-01-22)

ANNIS can be operated both as a centralized web-based service (for example with a powerful database server), but it is also possible to run it as stand-alone application on desktop systems.

2.3.4. Lucene-based query systems

Apache Lucene (Białecki et al. 2012) is a search library implemented in Java. Similar to CQP, Lucene is based on token streams, represented by fields of a document which can carry positional attributes. The field values can be retrieved via an inverted index. Since Lucene is optimized for fast retrieval of field values for large texts and is available as open-source "off-the-shelf" software, it has been used as a base for linguistic corpus query systems similar to CWB. Examples are the PELCRA search engine for the National Corpus of Polish (Pęzik 2011), BlackLab (Reynaert et al. 2014) or MTAS (Brouwer et al. 2017). While the first comes with its own query language, which is not designed to support flexible annotation layers, BlackLab and MTAS support the same query language that was introduced by CQP.

These Lucene-based query systems replicate the functionality of CQP (combined with different kind of user interfaces and other features that differentiate them), but they still rely on the same data model where annotations are either attached to tokens or spans. They do not integrate the functionality of treebanks into the same query system as ANNIS does. Like ANNIS, the query system KorAP (Bański et al. 2013) is intended as a collection of web-based tools and services that all together provide a corpus search platform for multi-layer corpora of different annotation types. Originally, it was proposed to use multiple back-ends based on different technologies for the different kind of annotations. Treebank-like annotations should have been implemented using a Neo4j-based back-end and spans using a Lucene-based back-end named Krill. Instead, the Krill corpus search system (Diewald and Margaretha 2016) now also implements the treebank-like annotations. It uses spans to describe the hierarchical entities and adds additional level information to these spans. KorAP is unique because it implements query languages of existing query systems, namely COSMAS II (Bodmer 1996), ANNIS and an extension of POLIQARP (Przepiórkowski et al. 2004). The AQL support is not complete, and a direct comparison of the performance of ANNIS and KorAP/Krill is hard due to subtle differences in the implementations. For example, Krill uses so-called foundries to separate the different sources of annotations (like different automatic annotation tools) and uses a slash in the annotation name to separate the foundry name from the annotation name. This syntax is illegal in the original AQL implementation, where namespaces separated with colons could be used for this distinction. Also, queries that are not specific enough (like a general search for all token), are rejected by KorAP/Krill and return no result. Additional problems occur when mapping corpora that have already been converted to ANNIS to KorAP/Krill, because the ingestion pipeline of Krill is specialized on processing annotation layers of the DeReKo corpus (Kupietz and Lüngen 2014), but is non-trivial to adapt to other types and formats of annotation.

2.4. Ziggurat: a proposal for a modular query system

CWB has been successfully used in several corpus linguistics projects, but its data model is limited to token annotations and spans. Therefore, the developers of CWB proposed a new data model named Ziggurat, whose implementation shall replace the current one (Evert and Hardie 2015b). A detailed description of Ziggurat is given in Evert and Hardie (2015a), and this section only describes some ideas that are relevant to the implementation of ANNIS. In Ziggurat, annotations are grouped into data layers, which are tabular-like and ordered by the position of the annotation. Data layers are specialized for the type of annotations they can contain, and they reference each other, building a hierarchy of data layers. There is a data layer type for segments of texts, another one for representing tree structures and one for storing general graphs. The special primary data layer is meant to represent the tokens and their annotations and is not referencing other data layers. It corresponds to the original CWB data model but is more flexible since the annotation values can have more complex types (like numbers and pointers to other annotation units in the same layer). Structural annotations or spans are encoded in segmentation data layers, which refer to the primary data layer by using an explicit range column. Data layers for trees also contain such a range column, but since their spans need to encode hierarchy information additionally, there is also a column with reference to its parent annotation unit and a column with reference to the following sibling. In contrast, the data layer for general graphs stores explicit pairs of positions that represent the edges. These positions can be annotation units from different data layers. Ziggurat takes a token-stream based data model with its efficient implementation and extends it to map more types of annotations while maintaining the original search approach, which the authors themselves describe as "brute-force" (Evert and Hardie 2015b, p. 23). Its possible implementation of the tree- and graph-like data layers is explicitly not based on a sophisticated graph-index (Evert and Hardie 2015a, p. 4). Still, some graph-related optimizations are planned, like using the pre-order of a tree annotation unit as its position (Evert and Hardie 2015a, p. 20).

The separation of the annotation layers in specialized data layers allows for optimized data storage and query strategies, but the specialization is based on the type of annotation, not the actual instantiation of the data. For example, tree structures can be very different: Some have limited height and many child nodes in average, while another extreme would be a tree with one child node in average, but thousands of nodes in a path. For new types of annotations, new optimized component implementations must be developed in Ziggurat. Since there is already a separation of the implementations for the different annotation layers, the optimization could be based on the actual instances of the data instead. It could also be preferable if different implementations are defined on the same data model and provide the same access functions. This is not the case for Ziggurat, where each different data layer type has different columns. A system which is exclusively based on graphs could provide the same access patterns (like efficiently finding reachable nodes from a starting point) for different component type implementations. Such a design would divide the complex problem of providing efficient strategies for searching linguistic annotations into the following sub-problems:

- mapping linguistic annotation concepts to graph components,
- implementing queries on annotations as queries on this graph, and
- providing efficient implementations for basic graph query operations for each component type.

Existing graph-based models for linguistic annotations are presented in the following chapter 3. A novel model for graphANNIS, which is based on these ideas, is introduced in Chapter 4. Its implementation is described in Chapter 5, additional optimizations of the implementation are presented in Chapter 6 and evaluated in Chapter 7.

3. Existing graph-based data models for representing and querying linguistic corpora

When building a system that can process corpora containing linguistic annotations, it is important to develop a data model which is suitable to represent the annotated data. In the specific case of a corpus search tool, an additional definition is needed for the structure and semantics of a search query and its results. The process of modeling the data can be implicit by choosing the appropriate internal data structures in the programming language of choice and by transforming data from external sources to and from this data model. It also can be an explicit step in the development process, where a model is created based on existing experiences with the data and the domain.

This chapter describes existing models that have been used to develop the legacy relational database implementation of ANNIS and are influential in the development of the new graph-based implementation. It also describes the query language that is implemented by ANNIS in more detail.

3.1. Describing linguistic corpora with Salt

Salt (Zipser 2009; Zipser and Romary 2010) is a meta-model for multi-layer corpora. It has been designed to be used in the conversion framework Pepper[1], and it maps every aspect of a multi-layer corpus to a labeled graph. It is not only an abstract model, but Salt is also an actual implementation of the model in the Java programming language. Since Salt has been used to map a huge variety of different kinds of linguistic corpora[2], it is very suited to be used as a base for the data model of graphANNIS. This section is meant to provide an overview about Salt and how it can be used to represent linguistic annotations. It is based on the ideas and descriptions of Salt of the original author Florian Zipser (Zipser 2009; Zipser and Romary 2010; Zipser 2012). Salt is under active development, and new versions are published at the Salt homepage[3].

[1] `http://corpus-tools.org/pepper/` (last accessed 2018-02-17)

[2] See for example `http://corpus-tools.org/pepper/knownModules.html` (last accessed 2017-11-01) for a list of supported annotation formats or `https://corpling.uis.georgetown.edu/annis-corpora/` (last accessed 2017-11-01) for an exemplary list of corpora.

[3] `http://corpus-tools.org/salt/` (last accessed 2017-10-25)

3.1.1. Model hierarchy

Salt is based on a labeled directed graph and defines classes for this basic structure. There are three different meta-models for different levels of abstraction. At the bottom, there is a meta-model for any kind of labeled directed graph called `GraphMM` (Zipser 2009, pp. 51–54). It uses an extended definition of a graph $G = (V, E, L, LAYER)$ where

- V is a set of nodes,
- $E \subseteq V \times V$ is a set of directed edges (with each edge defined as tuple $e = (v_1, v_2)$),
- L is a set of allowed labels (in Salt a label consists of a triple of strings, the namespace, name and its assigned value), and
- $LAYER$ is a set of layers.

Labels and layers can be assigned to both nodes and edges (the formal definition uses the special functions $label : (V \cup E) \rightarrow L$ and $layer : (V \cup E) \rightarrow LAYER$). This results in the following classes in the meta-model:

- `Graph` for instances of G
- `Node` for elements of V,
- `Edge` for elements of E,
- `Label` for elements of L, and
- `Layer` for elements of $LAYER$.

The meta-model additionally introduces a type hierarchy, for example, `Node`, `Edge`, and `Graph` inherit the common type `LabelableElement`[4]. This basic graph does not imply any linguistic semantics: it is just the definition of a labeled directed graph. On top of `GraphMM`, the `SaltCoreMM` meta-model is defined (Zipser 2009, pp. 55–57). It introduces certain concepts of linguistic annotation like `SAnnotation` and `SMetaAnnotation`. These are both extensions of `Label` and inherit all its properties, but by using different classes, it is possible to distinguish different kinds of linguistic annotations. `SaltCoreMM` also adds new classes as a replacement for the more general `Node` and `Edge` classes, named `SNode` and `SRelation`. The most extensive meta-model in the hierarchy is `SaltCommonMM` (Zipser 2009, pp. 57–63) which is defined on top of `SaltCoreMM`. It contains more elaborate kinds of linguistic annotation concepts, all of which are inherited from classes from `SaltCoreMM` and thus also the most basic `GraphMM`.

[4] This means that labels can be assigned to graphs as well, something that is not explicitly stated in the formal definition.

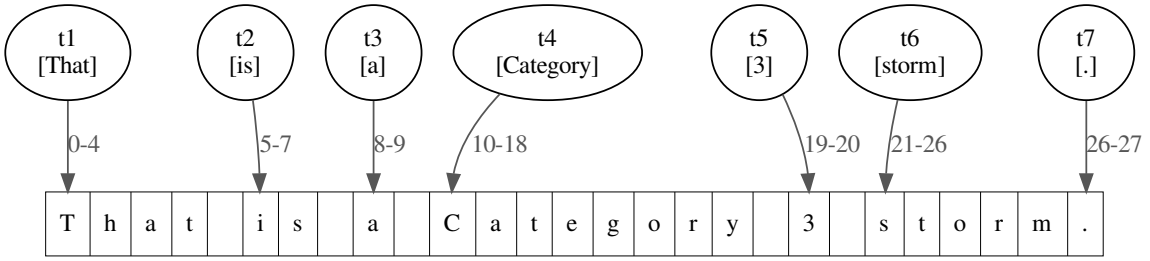


Figure 3.1.: Example of a textual data source and its connected tokens in Salt. The sample sentence "That is a Category 3 storm." is segmented into 7 tokens. The punctuation is included, but whitespace is not. Each token has an edge with the index of the first character in the text (inclusive) and the end character index (exclusive).

3.1.2. Linguistic annotation concepts represented in Salt

`SaltCommonMM` (from now on only referred to as Salt), divides corpora into sub-corpora and documents. Each document belongs to a corpus, and each corpus can be part of another corpus. Together this corpus structure forms the so-called *corpus graph*. The elements of the corpus graph can have meta-annotations attached. Corpora are divided into substructures for a various number of reasons, for example, because each document refers to a specific instance of coherent text, like a newspaper article. Also, grouping several documents into corpus hierarchies can be used as a way to express similarities of documents, like for example having the same author, the same place of origin or the same language. Dividing corpora into documents and grouping them into (sub-) corpora is an explicit decision made by the creators of the corpus and part of the modeling process (Odebrecht 2017, Section 2.6.1).

The actual linguistic annotations are expressed in the *document graph*. Each document has exactly one document graph assigned to it. Annotations inside a document graph are either expressed as a node, an edge or a label attached to either of these. Thus, the very existence of an edge between two nodes can express a certain linguistic feature. While there can be an edge between any element inside the corpus graph and inside each document graph, there is no possibility to add edges between elements of these two separated graphs. There is also no possibility to add edges between different document graphs: A document graph is a partition. This allows handling each document graph separately when processing queries or performing transformations. But it also means, that corpora which deal with phenomena that stretch over several logical documents, they must be modeled as one single document in Salt.

Salt assumes that the annotation graphs are connected with some kind of original *data source* that contain the actual act of speech. Currently, textual data sources (`STextualDS`) and media files (`SMedialDS`) are supported as data source types. A textual data source contains a text as string (represented by a sequence of characters each having a unique position in the sequence) and a media file data source refers to an audio or video file which has sequential time codes. Data sources can be segmented into tokens, which are modeled as explicit nodes of the type `SToken` and

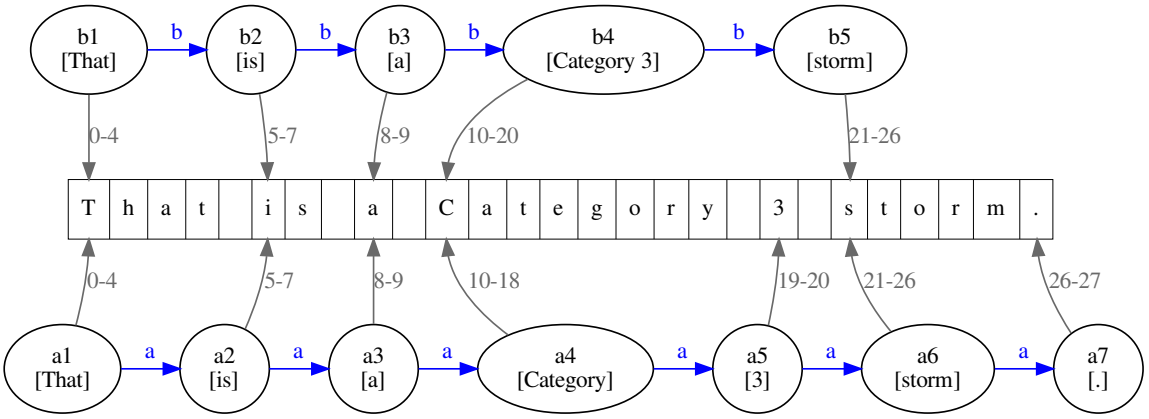


Figure 3.2.: Example for multiple segmentations of the same text. There are two segmentations here, "a" and "b". "a" is the same as in Figure 3.1 but "b" considers "Category 3" as one token and omits the punctuation. The explicit `SOrderRelation` edges are the blue ones.

which are connected to their data source with a special edge type (for example `STextualRelation` for textual data sources). This edge type contains the information which range of the data sequence is covered by the token as a label. Figure 3.1 shows an example textual data source and its tokenization. An advantage of this representation is that it is possible to preserve characters from the original text which might not be included in the tokenization, such as white-space or punctuation. Each document can have more than one data source, and each textual data source can be segmented in different ways. If there is only one segmentation, the order of the tokens is implicitly given by the indexes of the character range it covers. In case there are multiple segmentations (see Krause et al. (2012) and Odebrecht et al. (2017, pp. 698 ff.) for use-cases), explicit edges of the type `SOrderRelation` can be added to define the order of tokens (Zipser 2012, p. 17). An additional label on these edges can be used to name the different segmentations (see for example Figure 3.2). When the tokens are connected to other types of data sources, the edges also contain information about the range of the sequence on which a token is defined.

In addition to token annotations, Salt allows creating *hierarchical structures* using nodes of the type `SStructure`. These `SStructure` nodes can be connected with other structured nodes or tokens using edges having the `SDominanceRelation` type. Figure 3.3 shows an example of such a structure. Like the `STextualRelation`, a dominance relation implies text coverage. Thus every node covers all text ranges of all child nodes which are connected through a dominance relation.

Technically, it is possible to use an `SStructure` and `SDominanceRelation` to model non-hierarchical annotations on *ranges of text* as well. If the user wants to explicit express a non-hierarchical node, he or she can use the `SSpan` type instead. See Figure 3.4 for an example. Spans are connected to tokens with edges of the type `SSpanningRelation`, and there is an edge between the span and each covered token. An alternative would be to connect only the token at the beginning of the range and the one at the end. The benefit of explicit edges is that it is possible to model non-continuous ranges of text as well. Spans are used to model non-hierarchical text

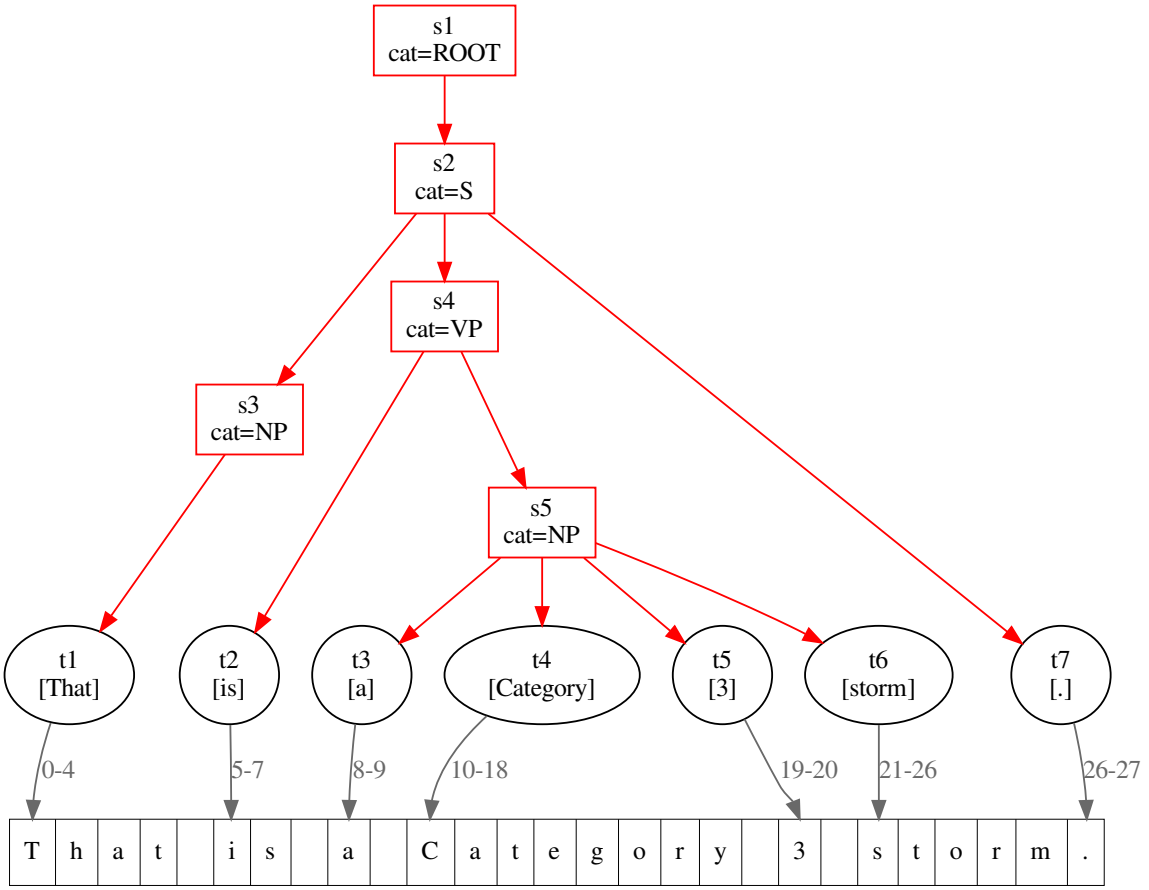


Figure 3.3.: Example for representation of structures in Salt. The red lines are instances of the `SDominanceRelation` type. Dominance relations imply text coverage, thus the node s4 with the `cat=VP` label covers the text range "is a Category 3 storm" because it is connected to other `SStructure` node s5 and the `SToken` t2, t3, t4,t and t6 that cover this text range.

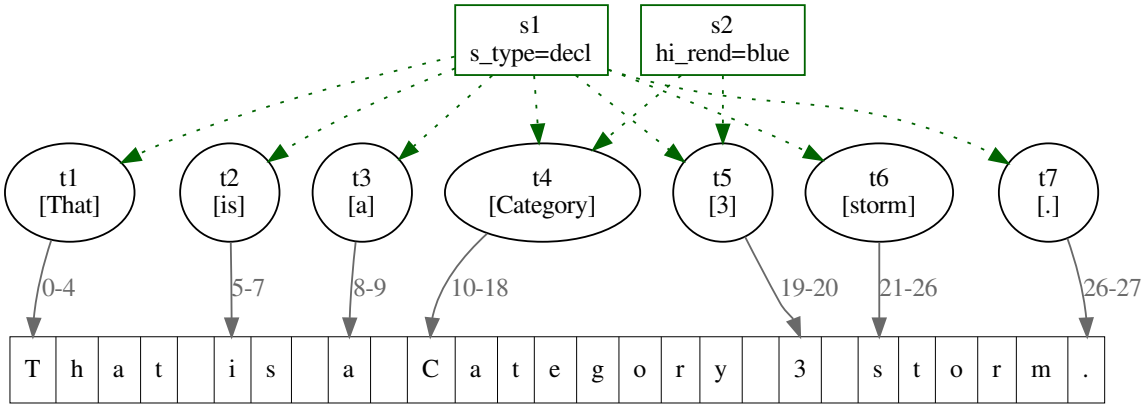


Figure 3.4.: Example for a span annotation in Salt. The green nodes are nodes of the type `SSpan`, and the dotted green lines are edges of the type `SSpanningRelation`. Span nodes have explicit edges to each covered token, instead of just having ones to the tokens at the start and end of the covered text range.

ranges, and they can only have outgoing edges to tokens, but it is possible to include them as target node of a hierarchical dominance relation.

Annotation nodes are connected to the text either indirectly or directly via edge types that implicate text coverage. Annotations can also be relations between nodes and these relations do not necessarily need to implicate text coverage. These kinds of annotations are expressed by using edges with the type `SPointingRelation`. Figure 3.5 gives an example of such an edge.

3.2. ANNIS Query Language (AQL)

This section introduces the ANNIS Query Language (AQL), which was developed as the query language for the existing ANNIS corpus search system and which is implemented by graphANNIS as well. The original design or any extension of AQL are explicitly not part of this work, and the following description of AQL is based on previous work (especially Rosenfeld (2010), Rosenfeld (2012), Krause and Zeldes (2016), and Zeldes (2016a)). AQL was inspired by other linguistic query systems, in particular, TIGERSearch (Lezius 2002).

3.2.1. Searching for linguistic concepts with a Domain Specific Language

In the previous section, the meta-model Salt was introduced. Salt is meant to represent and preserve linguistic annotation, and since it is based on labeled directed graphs (by inheriting the `GraphMM` meta-model), it would be possible to use an existing graph query language like Cypher (Robinson et al. 2013) directly with Salt. The user would need to understand how Salt maps linguistic annotations to a labeled graph to be able to write queries for the original linguistic phenomena. When using such a general

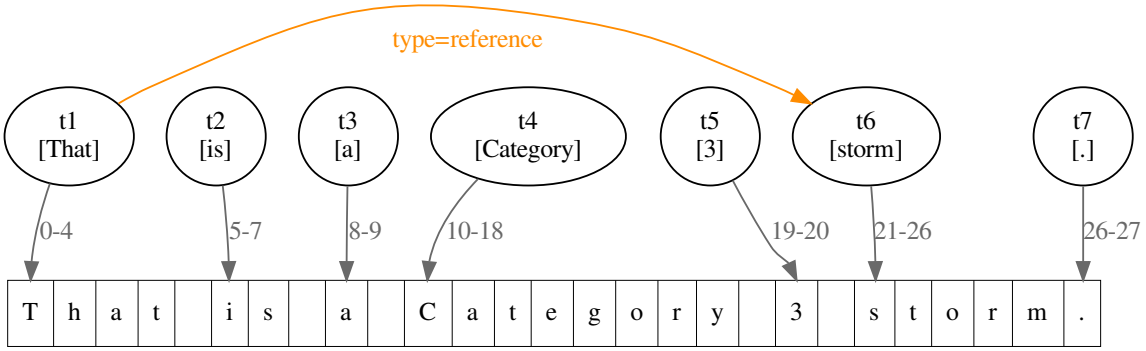


Figure 3.5.: Example for a pointing annotation in Salt. The pointing relation in orange does not implicate text coverage, and thus it is perfectly fine to connect tokens with each other. Edges of type `SPointingRelation` can connect any type of nodes and can have labels attached as an additional specification of the annotation.

graph query language with a different linguistic data model, the user would have to re-formulate its queries, despite using the same query language. The complexity of understanding the more general graph query language does not come with more flexibility in applying the query language to different corpora with different models.

AQL was designed as a DSL that uses concepts and terms that are familiar to linguists and are not necessarily referring to labeled graphs. While it would be possible to use general-purpose programming languages (GPLs) for executing corpus searches, DSLs have been used successfully in several existing linguistic query systems (see Section 2.3 for a description of some of them). Benefits of using a DSL have been described for example in Mernik et al. (2005):

> "DSLs trade generality for expressiveness in a limited domain. By providing notations and constructs tailored toward a particular application domain, they offer substantial gains in expressiveness and ease of use compared with GPLs for the domain in question, with corresponding gains in productivity and reduced maintenance costs. Also, by reducing the amount of domain and programming expertise needed, DSLs open up their application domain to a larger group of software developers compared to GPLs." (Mernik et al. 2005, p. 317)

Salt and AQL have been developed in parallel and corpora which are used in ANNIS have been first mapped to the uniform Salt meta-model and then converted to the ANNIS import format. Subgraph-extracting queries of the ANNIS API are in turn exported as XML-serialized Salt. Thus, every corpus available in ANNIS can be represented in Salt. ANNIS reduces the data model of Salt, and not all concepts of Salt can be searched in AQL, but AQL can be still interpreted as a query language on top of Salt. Additional to the goal of providing a DSL which is easy to use, it must be possible to provide an efficient implementation of AQL in a query system. Thus, the definition of the data model for the query language can be different from the one used by possible implementations. Implementations might choose to add additional

information which can not be directly queried in the query language (for example, for visualization purposes) or remove aspects of the data model, that are not needed by the query language, for performance reasons.

3.2.2. Searching for annotations in AQL

The most basic term in AQL is the annotation. This corresponds to a label in the labeled graph. A user can search for an annotation by its *name*, for example

```
pos
```

searches for all annotations with the name `pos` (which is a common abbreviation for "part of speech"). Additionally, a specific text *value* can be used as a search criterion. For example,

```
pos="NN"
```

will search for all annotations with the name "pos" and the exact value "NN" (which is used to describe certain types of nouns in some tag-sets like the Stuttgart-Tübingen-Tagset (STTS) (Schiller et al. 1999)). The only data type that can be used as annotation value is text, and no numeric data types are available. It is possible to use a regular expression in the search term for the value (but not the name) by using "/" instead of quotation marks:

```
pos=/N.*/
```

It is possible to further specify annotation names by using namespaces, for example when annotations from different sources with the same name are combined in a single corpus. Defining a namespace in a query is possible by using the ":" character, for example,

```
stts:pos="NN"
```

would search for the "pos" annotation of the "stts" namespace. Like in Salt, each annotation is attached to a node. A node can have multiple annotations attached to it, but there can not be more than one annotation per node having the same combination of namespace and name.

There are special annotation names in the query language that are reserved for more general concepts in the model. It is possible to search for any node by using the special annotation name `node`. Nodes might have some kind of internal identification, but it is not possible to specify the value of a node in a query. Thus, the search expression `node="something"` is invalid. Other specific kinds of nodes are tokens. Searching for the special annotation name `tok` will search for any token in the corpus. As with annotation values, you can search for tokens that cover a specific text by adding "=" to the search term:

```
tok="storm"
```

This expression can be expressed more compact by omitting the `tok=` part:

```
"storm"
```

The compact form will search the covered text of all segmentation nodes if a corpus has multiple segmentations. A user can specify which segmentation to search on by explicitly using the name of the segmentation as annotation name or by using `tok=` to only search for unnamed tokens. Regular expression search is also possible by using "/" instead of quotation marks. Unlike Salt, AQL does not have the concept of a textual data source, and it is therefore not possible to do a regular expression search over text that is covered by more than one token. Instead, one would search for different tokens and define how these tokens are connected to each other.

3.2.3. Combining terms with operators

It is possible to combine multiple terms in one query by separating these terms with the symbol "&". For example,

```
cat="S" & "storm" & #1 >* #2
```

searches for the terms `cat="S"` and `"storm"`, generates the Cartesian product of both results and then filters it by using the predicate `#1 >* #2`. This example predicate is composed of the binary operator `>*` and its two operands `#1` and `#2`. Binary operators have two operands, a left-hand side (LHS) and a right-hand side (RHS). In this example `#1` is the LHS and `#2` is the RHS. Each annotation search term is implicitly numbered and thus `cat="S"` can be referenced as first term `#1` and `"storm"` as second term `#2`. The operator `>*` only includes pairs of annotations that are connected by a path of dominance edges of arbitrary length. It is possible to use a more convenient AQL syntax and abbreviate queries by directly writing operators between two terms. For example, the above query could also be written as

```
cat="S" >* "storm"
```

AQL contains numerous binary and unary operators that express constraints on linguistic phenomena. These operators are defined on a graph-based data model, but their semantics are not primarily defined by graph operators but by linguistic annotation concepts. New operators can be added to AQL when needed by users. An up to date list of available operators can be found in the most recent version of the ANNIS user guide (Zeldes 2016a). Following, some operators that are currently available in AQL are described, but this list is not complete and only covers operators that are supported in graphANNIS.

Pointing relation operator "`->type`"

Pointing relations are relations between any type of annotations and correspond to edges of type `SPointingRelation` in Salt. They are explicitly typed. Thus, a relation must have a name. The pointing relation operator is written as `->type`, where "type" is the name of its type. This form is without a range parameter and corresponds to a single edge in the annotation graph. A range parameter can be added to the operator after the type name (possibly delimited with either a space or comma character). In

this case, the operator does not define a single edge but a path of edges of the same given type. A range definition can be either:

- `*` for a path of edges with this type of any length (for example `pos=/P.*/ & pos=/V.FIN/ & #1 ->dep* #2`[5]) or
- `m,n` where m is the minimal length of the path and n is the maximal length (for example `pos=/P.*/ & pos=/V.FIN/ & #1 ->dep,2,5 #2`).

Pointing relation components of a named type are not allowed to have cycles. The combination of several pointing relations with different named types can contain circles, and that is why the name of the type has to be given in the operator definition. For pointing relation queries without a range parameter, it can be defined that the relation must have an edge annotation as a constraint. The edge annotation definition is written in square brackets after the type, and the syntax is the same as for node annotations, for example, `pos=/P.*/ & pos=/V.FIN/ & #1 ->dep[func="sbj"] #2`.

Dominance operator ">"

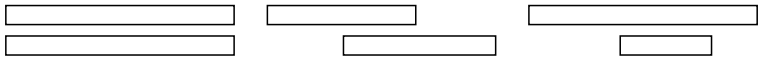
The dominance operator is similar to the pointing relation operator, but it corresponds to relations of the Salt type `SDominanceRelation`. Dominance relations are typed too, but the declaration of the type is optional for the dominance operator. Thus, the combination of all named components of dominance relations must still be cycle-free. As the pointing operator, paths of unspecified length can be expressed with `>*` (or `>type*` for selecting a dominance relation of a specific type) and a ranged length path can be specified with `>m,n`. Again, edge annotations can be given in square brackets for the single-edge variant of the operator: `>[func="SB"]`.

Precedence operator "."

A precedence relationship is defined over the stream of tokens. Two tokens are precedent if the LHS of the operator is located directly before the token defined by the RHS. For example, in the tokenized sentence "[That] [is] [a] [Category] [3] [storm]" the token "is" precedes the token "a" and the corresponding AQL query would be `"is" . "a"`. The precedence operator allows the same range argument as the pointing relation operator, with a `.*` marking an arbitrarily long distance between two tokens and `.m,n` marks a specified range $m..n$. Due to performance concerns, the legacy ANNIS implementations limited the maximal distance for the `.*` operator to 50 (while still allowing explicit ranges that define a larger distance). If a corpus uses multiple segmentations (for example by adding explicit `SOrderRelation` edges), the typed form `.type` of the operator can be used to specify the name of the segmentation.

Precedence is not only defined for tokens directly but also for all other annotation nodes since they are always directly or indirectly connected to a set of covered tokens. For such non-tokens, the right-most covered token is used as anchor point if the annotation is the LHS of the precedence operator and the left-most token for annotations on the RHS of the operator. For example, in Figure 3.3 the node s3

[5] The underline is only inserted for illustration purposes.



(a) same text `_=_` (b) overlap `_o_` (c) inclusion `_i_`

Figure 3.6.: Example spans for the different text coverage operators. The top span is the LHS, and the bottom span is the RHS.

with the NP annotation is precedent with distance 1 to the node s4 with the VP annotation because the right-most covered token of s3 ("That") is precedent to the left-most covered token of s4 ("is"). The typed form of the precedence operator can only be used to compare tokens which are part of the `SOrderRelation` directly, not to compare non-token nodes.

Text coverage operators "_=_", "_o_" and "_i_"

It was already described that each annotation node covers a specific set of tokens (which in turn cover a specific part of the original textual data source). The text coverage operators allow comparing these sets which each other. For the same text operator `_=_` the set of covered tokens must be equal, for the overlap operator `_o_` it is sufficient that there is any non-empty intersection of the two sets and the inclusion operator `_i_` filters annotation nodes where all covered tokens of the RHS are contained in the set of the LHS. An example for the different text coverage operators is given in Figure 3.6. These operators cannot be parameterized and are always defined on the untyped token layer and not on any named segmentation.

3.2.4. Searching for meta-annotations

Similar to Salt, AQL allows structuring linguistic corpora into documents. Each document belongs to exactly one corpus, and a corpus is a collection of documents. In AQL, all nodes of a specific result must be a member of the same document. Documents can have annotations which consist of name-value pairs like the regular annotations. It is possible to filter the results to only include documents which have certain annotation values by adding special terms which begin with the reserved name `meta::`. For example, in order to filter by documents having the annotation with the name "author" and the value "Jon Doe", you can add the term

```
meta::author="Jon Doe"
```

to an existing query. A complete example would be searching all mentions of the word "storm" by the author Jon Doe:

```
tok="storm" & meta::author="Jon Doe"
```

As with normal annotations, meta-annotations can have namespaces which are separated with a colon when defined in the query.

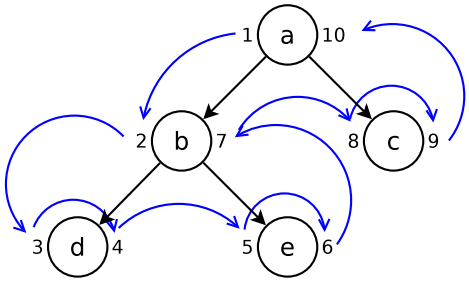


Figure 3.7.: Example tree with pre-order (numbers on the left side) and post-order (numbers on the right side). The blue lines trace the traversal order.

3.3. Modeling corpora in a relational database

Beginning with the second major version of ANNIS, it uses the relational database management system (DBMS) PostgreSQL to store the corpus data and to execute queries which are formulated in AQL. This relational database implementation (called relANNIS when it is important to distinguish it) was originally developed by Viktor Rosenfeld and is described in detail in Rosenfeld (2010). In this section, the basic ideas and problems of the relational database implementation are described and discussed.

3.3.1. Pre-/post-order encoding of reachability

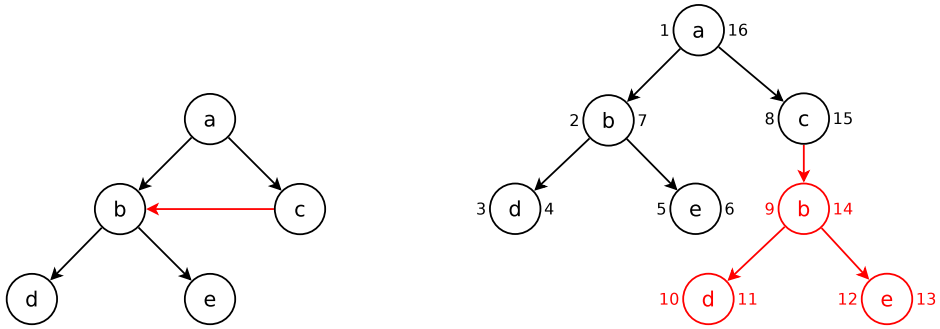
In Grust et al. (2004), pre- and post-order encoding was used to accelerate XPath queries of XML documents stored in a relational database. Since linguistic annotations are often expressed as a tree, and the AQL operators have similar functionality as XPath axis selectors, the relANNIS implementation uses the same approach to encode reachability of nodes. The tree is traversed depth-first, and the child-nodes are visited from left-to-right. Each time a node is entered the pre-order is assigned using a counter, and the counter is incremented by one. The post-order value is assigned when all children of a node have been visited.

RelANNIS uses the so-called "stretched pre-/post-order" or "combined pre-/post-order" where both the pre- and the post-order value use the same counter. Order values for an exemplary tree are given in Figure 3.7. By using the pre-/post-order value, it is easy to determine whether a node is a descendant of another node. A node w is a descendant of a node v if:

$$descendant(v, w) \Leftrightarrow pre(v) < pre(w) \text{ and } pre(w) < post(v) \tag{3.1}$$

This equation allows expressing a SQL query for finding all reachable nodes from a start-node by joining two columns from the LHS with two columns from the RHS. No recursive SQL or any other kind of extension to the relational algebra is needed. Fetching matching tuples for the SQL operator < can be supported by an index on these columns.

Unfortunately, pre- and post-order indexing only works for trees and not for DAGs. In order to work on DAGs, the concept must be expanded in a way that a node



(a) Additional edge between node c and b.

(b) Decomposed tree with pre-/post-order values.

Figure 3.8.: Example of how relANNIS encodes pre- and post-order for DAGs. An additional edge has been added to the example tree of Figure 3.7 between the nodes c and b (a). The DAG is decomposed into a tree by copying the descendants of node c (b). Copied nodes are drawn in red. As a result, the DAG has multiple order entries for the nodes that are reachable by several paths.

might have several pre- and post-order values instead of only one pair (Rosenfeld 2012, pp. 21 ff.). The basic idea is to copy sub-graphs, which are reachable by more than one path so that each sub-graph becomes a tree. Pre- and post-order entries are calculated for each of the generated tree copies, which leads to duplicated order entries. For example, if there is even only one additional edge in the graph of Figure 3.7 between node c and b, its descendant nodes b, d, and e are copied and assigned additional orders (see Figure 3.8). Depending on how many edges are added in comparison to the spanning tree of the DAG and where they are added, the duplication can be substantial. These duplicated entries must be filtered out after execution of the join because AQL only outputs each node annotation once in its result set. This adds additional processing overhead for corpora with duplicated entries, even if a specific result might not contain any duplicates.

3.3.2. Representing the graph in the relational database

When a user enters a query in AQL in the legacy ANNIS application, this query is translated to a SQL query, executed on the PostgreSQL database and the results are mapped to Salt. Thus, the original graph structure must be represented in the relational database, and it must be modeled in a way that allows efficient query execution. In the normalized schema, there are tables for the nodes, node annotations, edges and edge annotations. Figure 3.9 gives a graphical representation of this normalized version of the schema.

In order to describe the corpus graph structure and metadata, the `corpus` and `corpus_annotation` tables are used. An entry in the `corpus` table can be either a top-level corpus, a sub-corpus or a document. The hierarchic tree of documents is expressed by using a pre-/post-order encoding. Metadata annotations for documents and corpora are added to the `corpus_annotation` table.

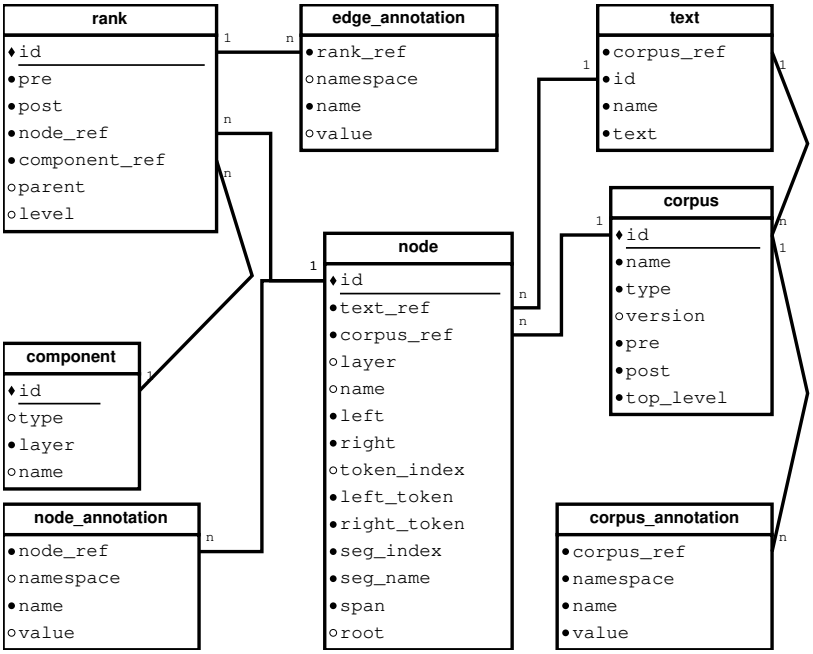


Figure 3.9.: Diagram of the relANNIS schema in normalized form. Only tables that represent the annotation and corpus graph are included. There are several more tables in the actual implementation, which are used for for example for configuration and references to binary data like media files.

The `node` table stores information about each annotation node. It is connected to the `corpus` and `text` tables which contain entries about each (sub-) corpus and each text. Thus, a node always belongs to exactly one document and one text. In order to speed up text coverage searches, the covered text range is explicitly encoded using the `left`/`right` and `left_token`/`right_token` columns. While the former ones describe the range of covered characters in the text, the latter ones refer to the covered tokens indexes. If a node is a token by itself, the `token_index` column is set to the index of the token in the chain of tokens for a specific textual data source (otherwise it is `NULL`). For nodes that are not tokens but segmentation nodes, the `seg_index` column is used instead. Additionally, the name of the segmentation chain for the node is given in the `seg_name` column. For either token or segmentation nodes, the `span` column is set to the spanned text value to make it possible to execute the `tok` searches without joining the `text` table. Annotations on nodes are encoded in the `node_annotation` table and consist of the textual representations of the namespace, name, and value of the annotation.

For expressing relations between nodes, the `component`, `rank` and `edge_annotation` tables are used. The `component` table lists all connected components of the annotation graph. A component has a `type` column which is used to distinguish pointing, dominance and spanning relations from each other. Additionally, the layer of the component is encoded in the `layer` column. Only edges belonging to the same type and layer can be part of the same component. The `rank` table is used to represent the edges of these components. Each `rank` entry contains an explicit reference to its parent rank entry via the `parent` column. Additionally, the connectivity is expressed

with the help of a pre- and post-order encoding.

3.3.3. Challenges of mapping graphs to a relational database

Using the pre-/post-order encoding approach allows to efficiently querying reachable nodes with only one join on two `rank` tables per operator. Unfortunately, a lot of information needed to execute an AQL query is encoded over multiple tables. This can result in joins of 9 tables even for simple queries, for example, queries which involve only two annotation nodes and one edge operator with a constraint on the edge annotation. In practice, these joins turned out to be very costly and thus a pre-joined materialized table named `facts` was created which combines the `node`, `node_annotation`, `component`, `rank` and `edge_annotation` tables into one large table. While this approach reduces the number of needed joins, it increases the size of the table and thus also the indexes. Joining the tables also results in almost duplicated rows which differ only in a few columns and are equal otherwise.

In the original schema, all corpora that are part of the database installation are inserted into the same `facts` table. This makes the size problem even worse since large corpora in the database can influence much smaller corpora. In order to make it easier for the DBMS to process the large `facts` table, it is partitioned by the top-level corpus a row belongs to. This is implemented by using a common parent table named `facts` and a child-table named `facts_<corpus ID>`, which inherits from the general `facts` table.[6] For queries that involve only one of the corpora, the SQL generation directly uses the child table name. Otherwise, PostgreSQL would use the statistics over all corpora instead of the more accurate statistics for a specific corpus. Depending on how different the corpora in the database are, this can make a huge difference. Whenever a query is executed over more than one corpus, the more general parent `facts` table is used, and PostgreSQL will determine which child tables to use on its own.

Another problem introduced by the materialized `facts` table are duplicated rows. This duplication requires filtering for distinct rows when only a subset of columns is queried. For queries that produce many results this filtering for unique rows is costly. Queries that should be fast to execute, because they do not have any operator, can take more time to execute compared to the normalized schema. In order to approach this problem, special columns where added to the `facts` table. These two boolean-typed columns `n_sample` and `na_sample` are only true for one row belonging to the same node or node annotation. Thus, it is possible to filter by these indexed columns instead of applying a unique filter on the output.

While the previous problems introduced by the `facts` table can be handled by various optimizations, there are inherent problems of mapping the graph-based data to a relational database. Statistical dependencies, which are introduced by columns that express a range (like the pre-/post-order columns and the ones for token coverage), are such a problem. These range columns are not statistically independent but the PostgreSQL query planner will assume this. Thus, the intermediate result size

[6] See `https://www.postgresql.org/docs/9.6/static/ddl-partitioning.html` (last accessed 2017-10-25) for a description how partitioning is implemented using inheritance in PostgreSQL 9.6.

estimation will be skewed, and PostgreSQL might tend to underestimate the number of tuples involved in a join and thus choose wrong join orders and join implementations. PostgreSQL has support for a proper range data type[7], but there is no actual support for statistics on these data-types at this time. As a countermeasure, a custom operator `^=^`, that replaces the "equals" operator `=`, was introduced for same coverage queries. It has the same semantics as the "equals" operator but has a constant selectivity of 0.995. In same coverage queries it is necessary to search for rows where the `left_token` and `right_token` columns of both sides are equal.

```
facts1.left_token = facts2.left_token AND
facts1.right_token = facts2.right_token
```

Since PostgreSQL assumes statistical independence the (accurate) selectivity for both column joins is multiplied, and as a result, the assumed result size is largely underestimated. By using the custom operator `^=^` for one of the sides, PostgreSQL will assume a constant selectivity for the `right_token` column join and make much better estimates.

```
facts1.left_token = facts2.left_token AND
facts1.right_token ^=^ facts2.right_token
```

Unfortunately, this approach is not generalizable for queries that use non-equal operators like `<` or `>`. Until the PostgreSQL query planner properly supports columns that are statistically dependent on each other, it will underestimate result sizes leading to bad join performance.

Another duplication problem is the indexing of reachability by using pre-/post-order encoding. This is the only graph indexing available for relANNIS, but it only works well in case if the annotation graph components are trees or have very few additional edges compared to the spanning tree of the component. Any extra edge will lead to duplicate entries in the `rank` table, and since the `facts` table is joined with `rank`, it can lead to a much larger number of rows per node in the `facts` table. There is no easy solution to this, other than allowing multiple ways of indexing graphs in relANNIS, which is not possible in the current SQL schema as it is and would also lead to a much more complicated SQL generation.

[7] `https://www.postgresql.org/docs/9.6/static/rangetypes.html` (last accessed 2017-10-25)

4. Graph-based data model for AQL searches

An important goal when designing graphANNIS was to avoid the mapping for graphs to the relational data model and the problems that this causes. As like Salt, the data model should be based on labeled graphs. In contrast to Salt, the data model should not be optimized to represent the annotation data but to allow efficient execution of AQL queries. This chapter describes the data model of graphANNIS, which is inspired by Salt and relANNIS, and shows how it can be used to represent linguistic annotations.

4.1. Basic concepts

Figure 4.1 gives an overview of the elements of the graphANNIS model, which is based on a directed graph (Cormen et al. 2009, p. 1168). Instead of partitioning the data into corpora like it was done in relANNIS, graphANNIS partitions the corpora into

- information about the nodes and node labels, and
- edges and edge label information which are partitioned again into components.

This allows separating the implementation for searching node annotation terms and the implementation of AQL operators. Operators are implemented by defining constraints on edges of the different components (see Section 5.6 for details).

In this model, each node is identified by a unique ID. Node labels consist of a namespace, a name, and a value and are connected to a node by its ID. No explicit

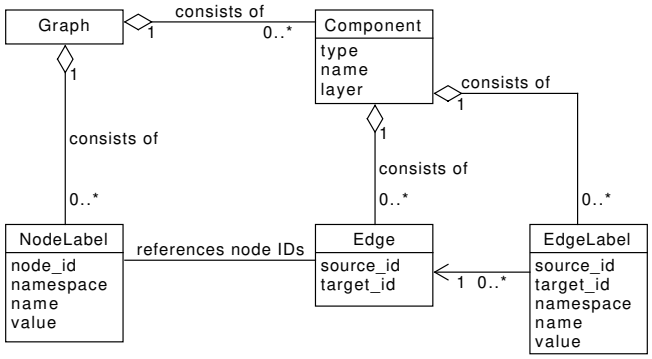


Figure 4.1.: Model of graphANNIS.

representation of nodes exists: If a node exists there must be at least one label for this node. There is a special label named "annis::node_name"[1] that can be applied to any node to mark its existence.

As relANNIS, graphANNIS has the concept of components for edges. A component has a type, a name, and a layer. It consists of edges and edge labels. Edges between two nodes are uniquely defined inside a component with the source and target node ID. There can not be more than one edge between the same two nodes inside the same component. Each edge can have multiple edge labels. In addition to the source and target node ID, edge labels also have namespaces, names and values. For one edge, only one edge label having the same namespace and name can exist. Graphs are the aggregation of node labels and edge components.

4.2. Representing linguistic annotations in graphANNIS

This section describes how different linguistic annotations are represented in graphANNIS, using the previous model of a labeled directed graph with typed and named edge components. In some cases, only the type of the component is used to identify it in the text, which means there can be only one component of this type in the graph.

4.2.1. Corpus and annotation graphs

GraphANNIS has two kinds of nodes:

- annotation graph nodes and
- corpus graph nodes.

They are both parts of the same graph structure but are distinguished by the special label "annis::node_type". It can either have the value "node" for nodes belonging to the annotation graph or "corpus" for nodes belonging to the corpus graph. Nodes that belong to a corpus graph are connected with edges that belong to a component of the type `PART_OF_SUBCORPUS`. The source node is always the node that is part of the (sub-) corpus, and the target node is the node which is higher in the corpus graph hierarchy. An example corpus graph is given in Figure 4.2. In this example, each annotation graph node belongs to exactly one document and the corpus graph is a tree. However, the data model allows to add an annotation node to several documents, and a document or sub-corpus can be part of several (sub-) corpora. In this regard, graphANNIS is more flexible than Salt.

4.2.2. Token

Since AQL has no concept of textual data sources, the leafs of the annotation graph in graphANNIS are the tokens. Tokens have a special label "annis::tok" which has the

[1] This is a fully qualified representation of the label name which includes the reserved namespace "annis".

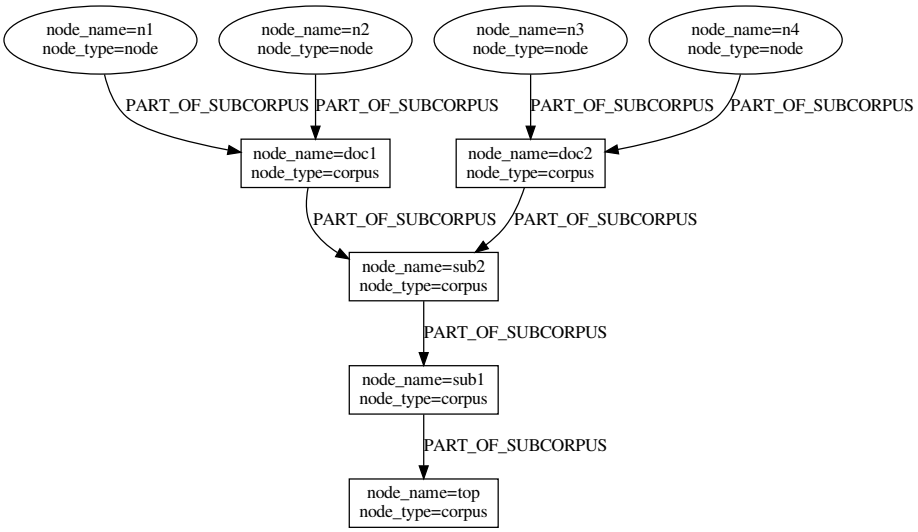


Figure 4.2.: Corpus graph representation in graphANNIS. This example corpus graph consists of a top-level corpus named "top", two sub-corpora ("sub1" and "sub2") and two documents ("doc1" and "doc2"). The annotation graph nodes have the type "node" while the others have the type "corpus". Edges of the corpus graph have the component type `PART_OF_SUBCORPUS`.

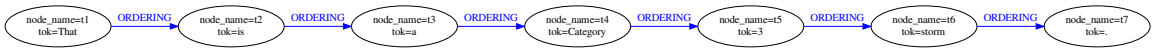


Figure 4.3.: Token representation in graphANNIS. The precedence of tokens is explicitly marked by the blue edges, which are part of the component with the type `ORDERING`. The special label "annis::tok" marks a node as a token and contains the spanned text value.

spanned text as its value. Additionally, tokens are connected with edges that belong to a component of type `ORDERING`. See Figure 4.3 for an example. The ordering edges are very similar to the explicit `SOrderRelation` edges in Salt, except that they are not obligatory but are needed to determine the order of the tokens in the absence of any character index. They also support multiple tokenizations because there can be more than one component of the type `ORDERING`. When there are multiple components with this type, the name of the component corresponds to the name of the tokenization and is empty for the default tokenization.

4.2.3. Spans

Spans are nodes that are not a token but cover a set of tokens. They also implicitly cover the original text that is covered by these tokens. GraphANNIS expresses these coverage relations by using explicit edges between the spans and each token it covers. These edges are part of a component of the type `COVERAGE`, and there are also edges in the inverse direction which are part of a component of the type `INVERSE_COVERAGE`. This allows to easily search for coverage in both directions. An example of how spans are represented is given in Figure 4.4. In addition to the coverage edges, there are

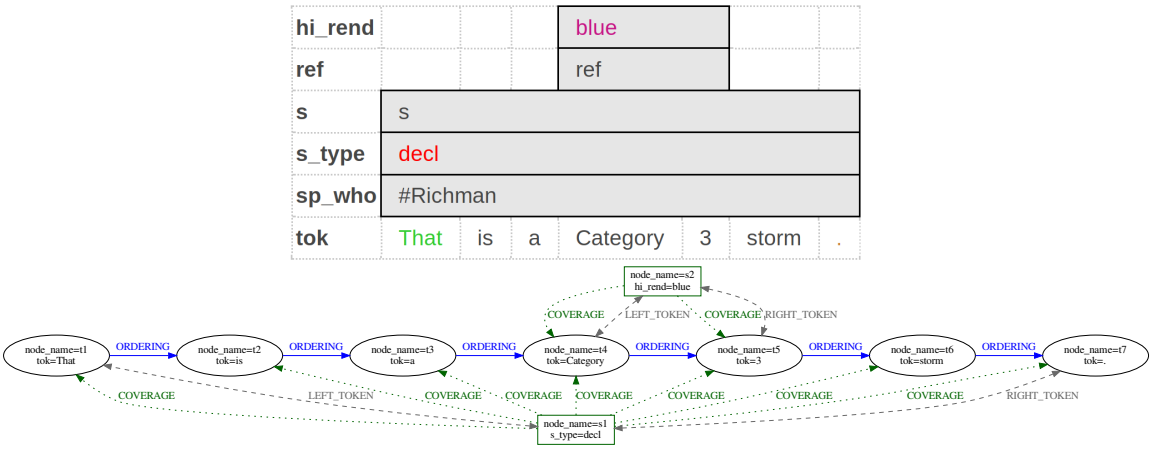


Figure 4.4.: Span representation in graphANNIS. The upper image shows an ANNIS front-end rendering of the example. In the lower image, the graph structure for the spans "hi_rend" and "s_type" is shown. There is an explicit edge from each span to each token it covers (represented by the dotted green edges with the component type `COVERAGE`). For each `COVERAGE` edge, there is also an edge of type `INVERSE_COVERAGE` with switched source and target nodes, but these edges have been omitted from the image in order to enhance readability. The edges in the `LEFT_TOKEN` and `RIGHT_TOKEN` component (represented by the dashed gray edges) explicitly mark the left-most and right-most covered token of the span.

also edges that mark the left-most and right-most covered tokens. These components of type `LEFT_TOKEN` or `RIGHT_TOKEN` allow providing faster implementations for some operators that deal with text coverage or precedence. While the coverage edges are similar to the `SSpanningRelation`, the left and right token edges are inspired from the two columns of the `node` table in relANNIS with the same name. Each node of the annotation graph that is not a token must have a left and right token edge because AQL implicitly requires all nodes to be connected to tokens. Tokens are also connected to non-token nodes with an inverse edge in the `LEFT_TOKEN` and `RIGHT_TOKEN` component. This allows for a faster lookup for all left- or right-aligned nodes of a token.

4.2.4. Dominance relations

While spans are used to describe non-hierarchical node structures, hierarchical structures like constituent trees are modeled using edges of the type `DOMINANCE`. An example of such a constituent tree is given in Figure 4.5. These edges can also have additional labels if the annotation scheme requires it. In contrast to Salt, where nodes have different types, in graphANNIS, only the edges are typed. Thus, `DOMINANCE` edges can be in theory added to any node. Since they imply text-coverage, these components should be non-cyclic. Also, graphANNIS explicitly adds edges for the left- and right-most covered tokens, while in Salt this is only implicitly given by the reachable tokens for a node.

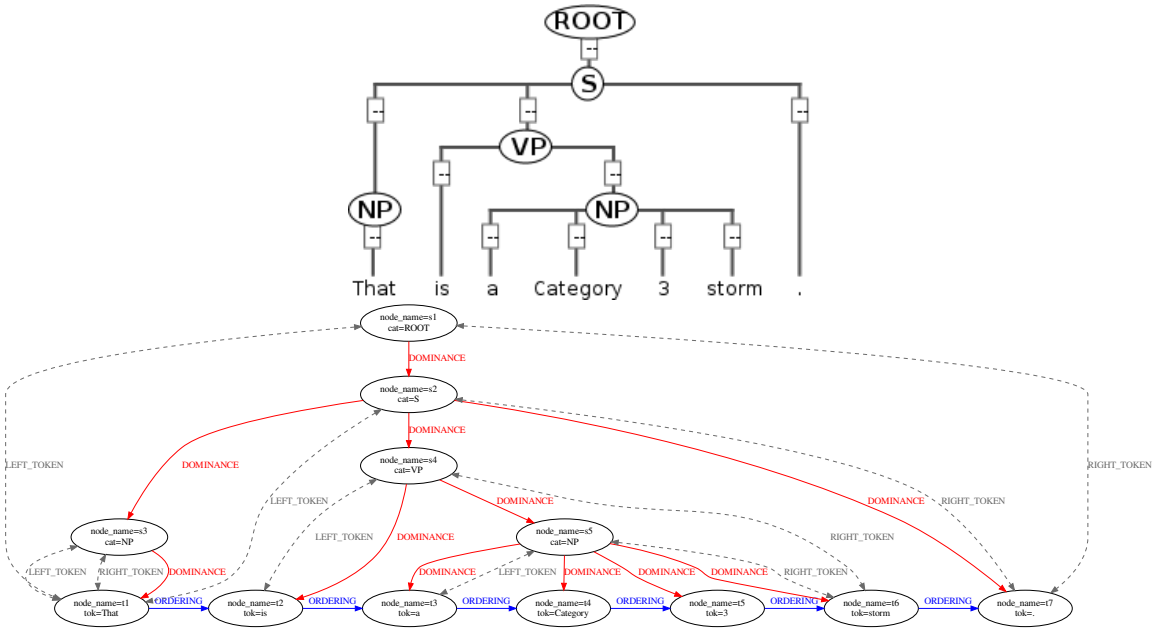


Figure 4.5.: Example for a constituent tree represented in graphANNIS. The red edges are of the type `DOMINANCE`. In this example, the edges do not have additional labels, but it is possible to add them if needed. Since these edges imply text coverage, the left and right token edges are inherited from child to parent nodes. For example, the node `s3` has the first token as the left-most covered token, and thus its parent node `s2` has the same left-most covered token node.

4.2.5. Pointing relations

For relations that are not implying text-coverage, edges of type `POINTING` can be used. These have the same semantics as the `SPointingRelation` of Salt and can have additional labels for expressing edge annotations. The type of a pointing relation in AQL corresponds to the name of the edge component in graphANNIS.

4.3. GraphANNIS data model and AQL

At the beginning of this section it is stressed, that graphANNIS is meant to be a model that makes it easy and efficient to implement AQL queries. Node annotation search is comparatively limited in what it can express and can be implemented relatively straight-forward. In contrast, implementation of the different operators as they have been described in Section 3.2.3 is more challenging. There are numerous operators with different semantics, and it is not desirable to keep special encoded and optimized information for each operator separately in the database. Instead, an operator should be implemented by combining several more basic graph component types.

Also, these operators have something in common: In addition to finding direct child nodes, they often need to find all reachable nodes from a given start node with some additional constraints like path length or valid edge types and labels. This means it

must be possible to implement a reachability query for a component very efficiently. That is why relANNIS uses the pre-/post-order encoding to query for reachable nodes without the need to traverse through the graph with recursive SQL.

But the components that represent the different aspects of linguistic annotation are very different from each other. The `COVERAGE` component only has paths of length 1 because there are not any hierarchies for spans. Also, `ORDERING` components can have very long paths (depending on the text length), but a node always has at most one outgoing edge. These different graph structures for different linguistic annotations have different optimal implementations for reachability queries. Separating graphs into components of different types and names allows exploiting these differences and provide a more optimal implementation. Still, these implementations are based on graphs and do not need a translation into a different model. For example, the token coverage of nodes is expressed as attributes of the node in relANNIS, which extends the data model of nodes. In graphANNIS, edges are used to encode the same information.

4.4. Extensions to the relational algebra to model AQL queries

GraphANNIS is based on a labeled directed graph, and thus it is not trivial to apply relational algebra on the graph itself. It is, however possible to describe the results of the query as relations. For example, the openCypher query language also operates on a very similar graph structure called property graph (Rodriguez and Neubauer 2010) and in "Formalising openCypher Graph Queries in Relational Algebra" the gap between graphs and relational algebra is bridged by stating that "openCypher queries take a property graph as their input, however the result of a query is not a graph, but a graph relation" (Marton et al. 2017, p. 184). OpenCypher adds new operators to the relational algebra, which use the graph as input and produce relations. Using relations to describe the results of the query is very similar to a definition of a result in graphANNIS, where each match is defined by a tuple of nodes and the matching node annotation. Since the edge information is not part of a graphANNIS result, it is sufficient to describe only the node labels as a relation. More formally, a node label relation is a set of tuples

$$N = \{(id, ns, name, val) \in \mathbb{N} \times D_S \times D_S \times D_S\} \tag{4.1}$$

where D_S is the domain of all strings, id is the global ID of the node, ns and $name$ represent the qualified label name of the match and val is its value. The relation can also be expressed as relational schema definition:

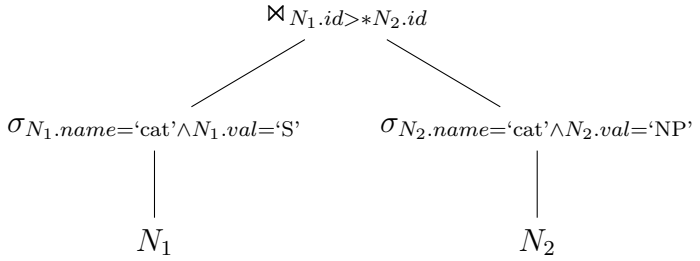
$$N(id : \mathbb{N}, ns : D_S, name : D_S, val : D_S) \tag{4.2}$$

A simple AQL query like `anno_ns:anno_name="value"` on a node label can be expressed as a selection on such a node relation:

$$\sigma_{N.ns=\text{'anno\_ns'} \wedge N.name=\text{'anno\_name'} \wedge N.val=\text{'value'}}(N) \tag{4.3}$$

Binary AQL operators that combine two node annotation searches are expressed as θ-joins $\bowtie_\theta$. In the original relational algebra definition, the operators that can be used as θ-condition are restricted to the basic mathematical comparison operators (Codd 1972). This is not sufficient for graphANNIS, where the conditions can be much more complex and involve properties of the input graph. Thus, we have to allow every AQL operator as a valid binary condition between two tuples involved in a θ-join. The AQL operators are only defined for the node ID attribute *id*. Using this extension allows writing the example query `cat="S" & cat="NP" & #1 >* #2` as relational algebra expression with two source relations N_1 and N_2 (for both annotation searches) and a join on the node IDs:

$$\bowtie_{N_1.id>*N_2.id} (\sigma_{N_1.name=\text{`cat'} \wedge N_1.val=\text{`S'}}(N_1), \sigma_{N_2.name=\text{`cat'} \wedge N_2.val=\text{`NP'}}(N_2)) \tag{4.4}$$



Aliasing the original N relation as N_1 and N_2 helps to uniquely identify the attributes in the join conditions without renaming the columns.

5. Graph-based implementation of AQL

This chapter describes the implementation of graphANNIS[1], which uses the graph-based data model for AQL searches proposed in Chapter 4. It will be evaluated with benchmarks in Chapter 7. First, the general architecture is described, followed by a description of the query execution workflow. Second, the components of the system that are relevant for the execution of queries are described in more detail. This includes storage concepts used for the different parts of the data model such as node annotations and edges, but also the implementation of the different AQL operators and joins.

5.1. Architecture

GraphANNIS is meant to replace parts of an existing complex software system, which has been used in production since several years and by numerous researchers. The current ANNIS application encompasses a front-end web application and a back-end service. The web application communicates with the service via a REST interface (Fielding 2000, pp. 76 ff.) and the service communicates with an existing PostgreSQL database via its network interface. See Figure 5.1 for an architectural overview how queries are processed in the current ANNIS system. The interfaces of all three components are network-based, but it is up to the administrator of the system to decide if each component is located on a different server or not.

The existing ANNIS system is mainly written in the Java programming language. ANNIS uses the Vaadin[2] framework as base for the web application. Vaadin itself uses Java Servlets[3], the Google Web Toolkit[4] and JavaScript. Some parts of the visualization in the front-end are directly written in JavaScript. The back-end is also written in Java and uses JDBC[5] to communicate with the database server. While the back-end is responsible for mapping between the different query languages and data models, the actual query execution is always performed by the PostgreSQL database.

[1] The software is available as Open-Source under `https://github.com/thomaskrause/graphANNIS`. Additionally, the specific release of graphANNIS which is used in this work is also archived in the research data repository Zenodo and can be accessed at `https://doi.org/10.5281/zenodo.1146565`.

[2] `https://vaadin.com/` (last accessed 2017-11-13)

[3] `http://www.oracle.com/technetwork/java/index-jsp-135475.html` (last accessed 2018-02-17)

[4] `http://www.gwtproject.org/` (last accessed 2018-02-17)

[5] `https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/` (last accessed 2017-11-13)

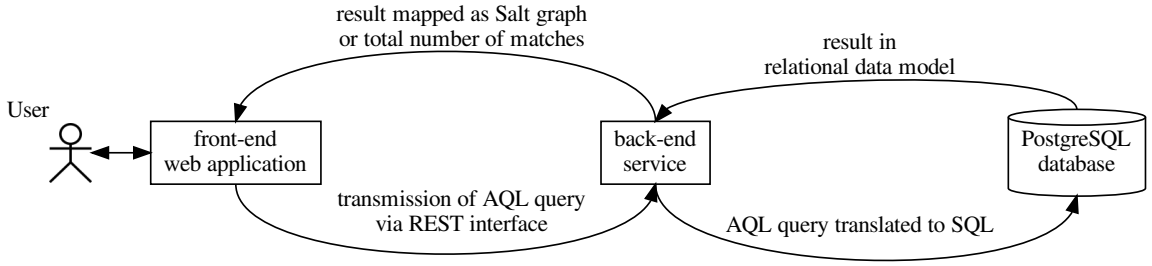


Figure 5.1.: Overview of the components of the current ANNIS system. AQL queries are entered by the user into the web application, sent to the back-end and then the back-end translates AQL to corresponding SQL statements. The result of the SQL queries are then mapped to a Salt graph representation in the service and are sent to the web application where it can be visualized.

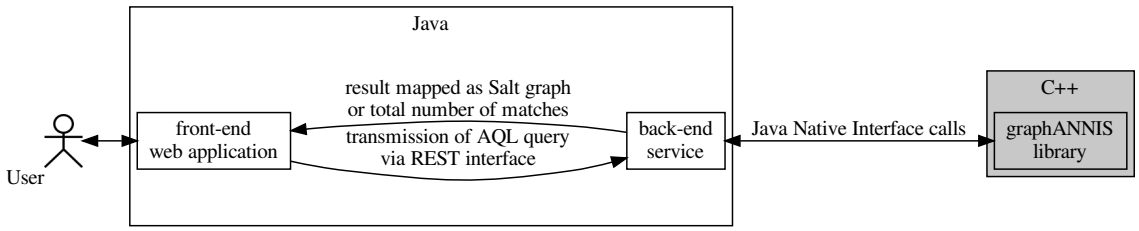


Figure 5.2.: Overview of the components of the new ANNIS system. Unlike the legacy ANNIS system (Figure 5.1) there is no outgoing network connections from the back-end service. Instead, the service uses graphANNIS as embedded library.

GraphANNIS will replace the PostgreSQL database as query processing engine. In the new system, the back-end service is calling functions of graphANNIS directly. This means the back-end service uses graphANNIS as a software library and not as an independent application like a network-based RDBMS.[6] Figure 5.2 gives an overview of the components of the new ANNIS system. The connection between the Java-based back-end and the C++-based graphANNIS library is implemented using the JavaCPP library[7], which automatically generates a Java Native Interface binding from C++ classes.

GraphANNIS is written in C++14 and uses several external libraries for some of its functionality. Notable, these include

- Google C++ B-tree[8] for an efficient and cache-sensitive B-tree implementation,
- Google RE2 regular expression library[9],

[6] A consequence of this approach is, that the back-end service and graphANNIS are always running on the same computer. Thus, load distribution over a network of computer systems would be a feature of the back-end service or a proxy system that combines several back-ends.

[7] `https://github.com/bytedeco/javacpp` (last accessed 2017-10-25)

[8] `https://code.google.com/archive/p/cpp-btree/` (last accessed 2017-11-13)

[9] `https://github.com/google/re2` (last accessed 2017-11-13)

- Boost[10] for a flat map implementation and various helper functions,
- Vc SIMD support library[11], and
- Cereal serialization library[12].

5.2. Query execution workflow and query representation

This section introduces the general workflow for executing AQL queries and describes how execution plans are represented. Executing an AQL query on the annotation graph returns a set of matches. Only unique matches are included in this result set. A single match is an array of triples of the form $(id, ns, name)$. Each triple contains the node ID and the qualified name (namespace and name) of the matching label. Information about the edges is not part of the result set. Matches for the example query `cat="S" & "storm" & #1 >* #2` contain two triples, because the query defines two annotation searches (`cat="S"` and `"storm"`).

$$[(101, \text{“tiger”}, \text{“cat”}), (2034, \text{“annis”}, \text{“node\_type”})] \tag{5.1}$$

This example matches the nodes with the ID 101 and 2034. In the example query, only annotations with the name “cat” can match the first element of the tuple. Since the namespace is unspecified in the query, a label with any namespace can match the condition: In this example this is the “tiger” namespace. The second element of the array matches the node itself, which corresponds to the special label “annis:node_type” (all nodes of the annotation graph have this label, see Section 4.2.1). A match X is different from another match Y, if any triple $X[i]$ is different from its counterpart $Y[i]$ at the same position i. A triple is different, if any of the three elements differ. For example, even if the nodes of the match $[(101, \text{“tiger”}, \text{“cat”})]$ and $[(101, \text{“example”}, \text{“cat”})]$ are equal, both matches are included in the same result set because the namespace of the label differs.

GraphANNIS uses so-called iterators (Garcia-Molina et al. 2000, pp. 261 ff.) to produce the elements of the result set iteratively. An iterator provides a `next()` function, which returns either

- the next array of triples, or
- an empty result if there are no more matches in the result set.

In addition, the `reset()` function will reset an iterator, so it restarts output of all matches from the beginning. Different iterators are responsible for fetching different parts of the query. They do not necessarily return an array for all nodes of the query, but only for the subset they are responsible for. Iterators can be combined into a new iterator which emits the concatenation of the arrays of its sub-iterators.

[10] `http://www.boost.org/` (last accessed 2017-11-13)
[11] `https://github.com/VcDevel/Vc` (last accessed 2017-11-13)
[12] `https://uscilab.github.io/cereal/` (last accessed 2017-11-13)

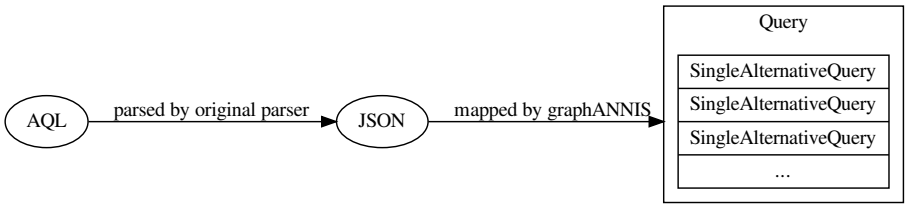


Figure 5.3.: Query parsing workflow. GraphANNIS does not parse the AQL query on its own, but receives a JavaScript Object Notation (JSON) representation from the parser. This JSON is then mapped to an object of the type `Query`. A `Query` object consists of several conjunctions, each of them represented by an instance of the class `SingleAlternativeQuery`.

Queries are given as string in the JSON format to the Application Programming Interface (API) of graphANNIS. See Figure 5.3 for an overview of the parsing workflow. This JSON is generated by the same Java-based parser that is used by the legacy ANNIS implementation, but limits the accepted AQL to only include operators implemented in graphANNIS. An AQL query can be interpreted as a boolean formula, where each term represents a condition, and a match is included in the result set if the formula evaluates to true for this match. AQL allows to connect the terms with "and" (`&`) and "or" (`|`). The AQL query which is provided by the parser is normalized to disjunctive normal form (DNF). In DNF, a boolean formula consists of a disjunction of conjunctions (Cormen et al. 2009, pp. 1083 ff.):

$$(t_1 \wedge t_2 \wedge \dots) \vee (t_i \wedge t_{i+1} \wedge \dots) \vee \dots \tag{5.2}$$

GraphANNIS receives the JSON representation and translates it into a data structure named `Query`, which includes the list of conjunctions and where each conjunction is represented by the class `SingleAlternativeQuery`. `Query` implements an iterator which iterates over all conjunctions once and filters the output to only include unique results. The `SingleAlternativeQuery` represents the logical representation of a single conjunction and stores the definition of its terms. Terms are grouped into two lists, one for the node annotation search definitions and one for the AQL operator definitions.

`SingleAlternativeQuery` implements an iterator for the result set by creating an execution plan when the `next(…)` function is called the first time. Then, `next(…)` is called on the execution plan and the result is returned. Subsequent calls to `next(…)` on the `SingleAlternativeQuery` are directly forwarded to the `next(…)` function of the execution plan. An execution plan is used as a framework to organize and represent the different steps that are needed for computing the results of a specific query. It is a tree structure composed of instances of the class `ExecutionNode`, which are also implemented as iterators. Each execution node is responsible for a part of the query. In contrast to the logical representation of the query with the `Query` and `SingleAlternativeQuery` classes, an execution plan is a so-called physical plan (Garcia-Molina et al. 2000, pp. 238 f.). See Figure 5.4 for an overview of the classes involved in the execution plan and Figure 5.5 for a detailed description of an example workflow. Each `ExecutionNode` has a type, which can be

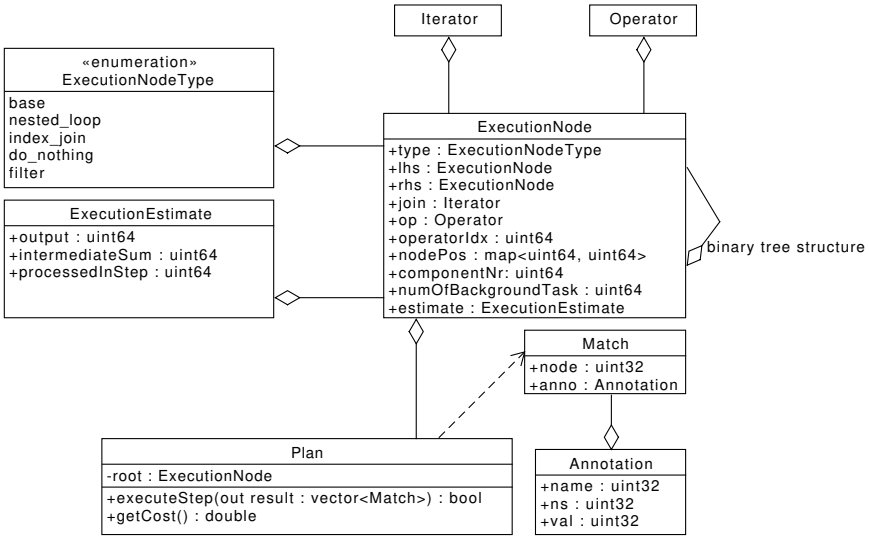


Figure 5.4.: Class diagram of the `Plan`, `ExecutionNode` and related classes. The `Plan` has a reference to the root node of the tree of execution nodes, and they reference the iterators and operators needed to execute the query. Each `ExecutionNode` also contains result estimations used for generating optimized plans.

- `base` for annotation search nodes (see Section 5.4 for all classes of this type),
- `nested_loop` or `index_join` for joins that combine a left-hand side (LHS) and right-hand side (RHS),
- `filter` for filtering combinations of a LHS and RHS but not creating new elements, or
- `do_nothing` for execution nodes that are known not to produce any results.

An execution node that is not of the `base` type refers to its child execution nodes with the `lhs` and `rhs` fields. For joins and filter operations, there is also a reference to the AQL operator implementation used in the field `op` (see Section 5.6 for a description of all implementations). Execution nodes represent the tree structure of the execution plan, but they do not implement the physical operations themselves. Instead, they refer to the actual iterator implementation with the `join` field. Despite its name, this field can reference an actual join, a filter implementation, or an annotation search iterator. In addition to the information on how to execute the different parts of the query, the execution plan also contains information about the estimated result sizes of each node which can be used to estimate the cost of the overall plan.

Relational algebra has been used in Section 4.4 to express the logical query plan for AQL, but it can also be used to describe the physical query plan tree. For example, the query `/A.*/ & pos="NN" & #1 _=_ #2` for finding nouns which start with the letter “A” could be implemented by the query plan:

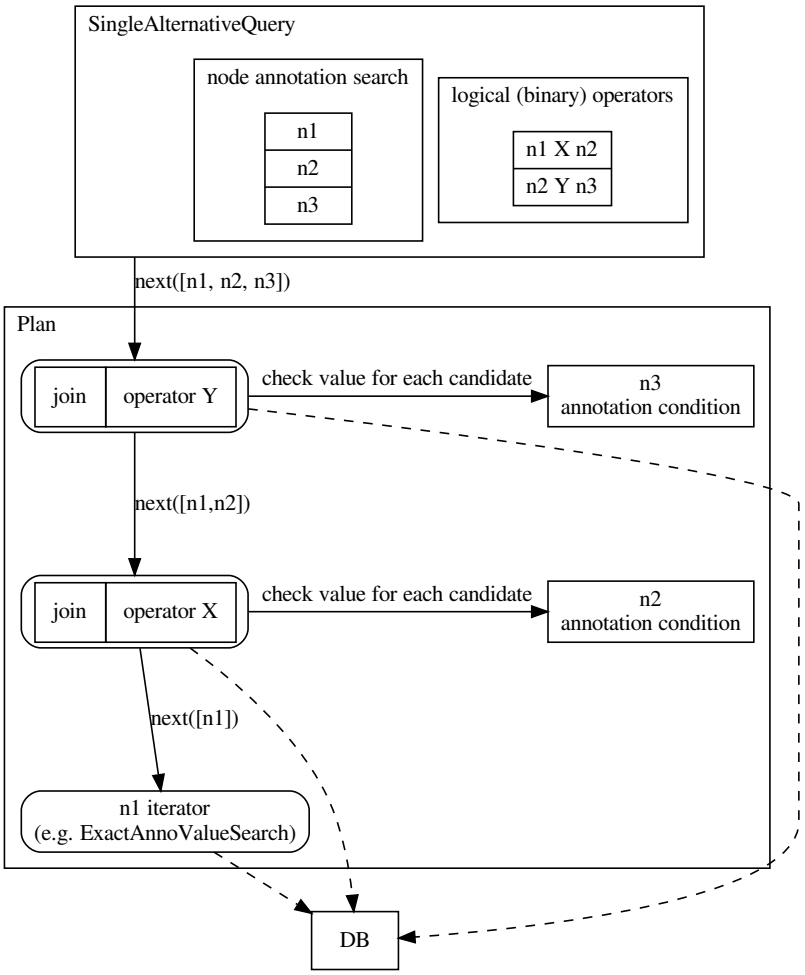
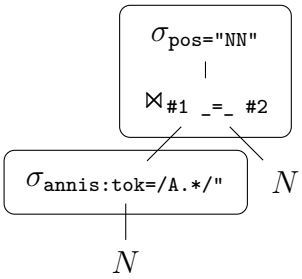
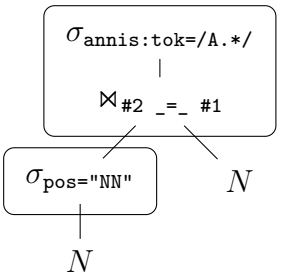


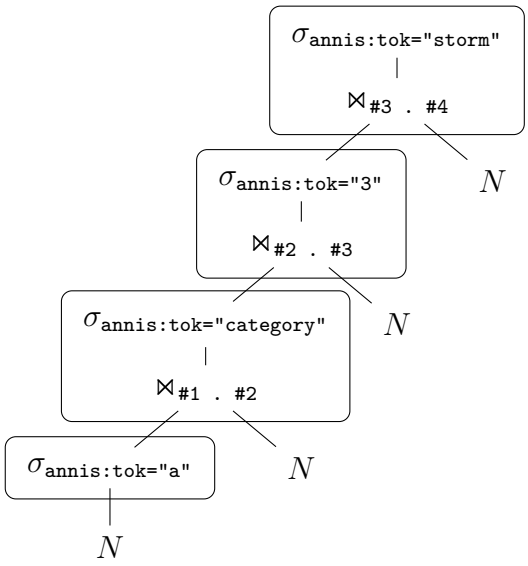
Figure 5.5.: Query execution workflow example. Each `SingleAlternativeQuery` creates an execution plan when its `next(…)` function is called the first time. In this example, the conjunction consists of three nodes ($n1$, $n2$ and $n3$) and two AQL operator definitions (X and Y). Each of the two operators results in a join execution node in the plan. These joins fetch results from their child execution nodes and apply the operator on these nodes. As described in Section 3.2.3, AQL operators have an annotation node search reference as left-hand side (LHS) and right-hand side (RHS). This is reflected in the execution plan, where the join nodes also have a LHS and RHS execution node. The joins in the example call the `next(…)` function of their LHS to get the next sub-array of the match. Then, they find candidate nodes which fulfill the associated operator definition by calls to the database (indicated by the dotted lines) and check each candidate node if it has a label that fulfills the annotation condition for the RHS node annotation search. A join then returns the concatenation of its LHS result array with the new RHS match triple. The child node in the execution plan ("n1 iterator") is an iterator that finds all match triples that fulfill the node annotation search condition by querying the database for matching labels.



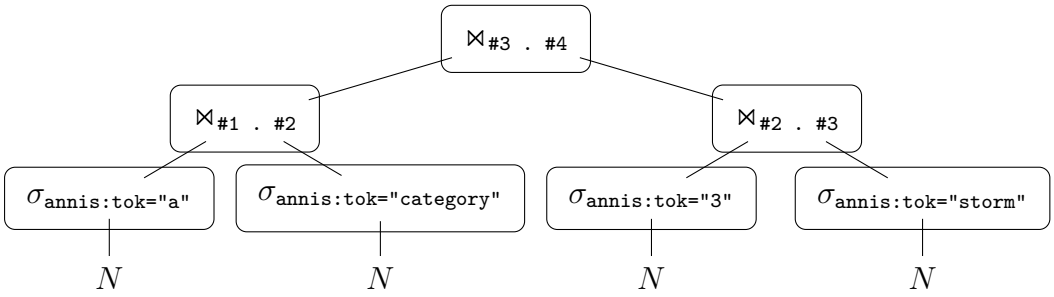
Execution nodes in the plan are indicated with boxes. The join $\bowtie$ combines its LHS node annotation search with the RHS using the logical operator `_=_`.[13] Since the same text coverage operator `_=_` is commutative, a query plan generator could decide to switch the LHS and RHS. This would result in a different physical plan for the same query:



The process of determining the best physical plan will be explained in more detail in Chapter 6. The more terms a query has, the more possible alternative plans can be generated. For example, possible query plans for `"a" & "category" & "3" & "storm" & #1 . #2 & #2 . #3 & #3 . #4` (which searches for the phrase "a category 3 storm") are the following ones:



[13] The join node $\bowtie_{\texttt{\#1 \_=\_ \#2}}$ and the selection $\sigma_{\texttt{pos="NN"}}$ are written as two nodes in the relational algebra plan to follow traditional notation, but graphANNIS can use a single `ExecutionNode` of the type `index_join` to implement both of them (see Section 5.7 for details). This is why the two relational algebra nodes are part of the same box.



5.3. String representation and storage

While the previous sections describe the general architecture and workflow, the following sections present the different components to implement the data storage and query execution in more detail. This section describes the component that is responsible for storing strings in the database. Strings are an essential part of any corpus query system because they are the most basic things that can be searched for. Labels on the nodes and edges consists of three strings: a namespace, a name, and a value. Also, being able to efficiently execute regular expression searches on strings is important, because regular expressions can be a powerful tool for linguistic pattern search. When deciding how to represent strings in the database, it is important to consider the overall impact of this decision on the whole system.

GraphANNIS is main memory-based and main memory is usually more limited than disk space. It is important that the data representation in graphANNIS is as compact as possible. One way of saving memory space is by using compact representations for duplicate values. Strings in graphANNIS can be duplicates for several reasons:

- When the strings represent words of natural language, it can be expected that the distribution of occurrence frequencies for unique strings in a corpus follows Zipf's law (Baroni 2009).

- Namespaces-name combinations of labels are the same for a number of different labels, where only the label value differs.

- Annotations in linguistic corpora are often based on fixed annotation schemes, where the value of an annotation is part of a pre-defined set of allowed values.

In order to increase the efficiency for representing such duplicates, graphANNIS uses a central dictionary for all strings that are used as annotation names or values. This dictionary is implemented in the `StringStorage` class (see Figure 5.6 for its class diagram). All strings in the dictionary are UTF-8 encoded. The key of the dictionary is a 32 bit integer, so storing a reference to a string takes 4 bytes. To achieve a faster lookup, the `StringStorage` class contains both a map from the ID to the string and an inverse map from the string to its ID. Storing the entry in the dictionary takes $4 \cdot 2 = 8$ bytes in addition to twice the length of the string (plus the overhead from the container classes). Each reference in a label to a string will use 4 additional bytes. Thus, only strings that are larger than 4 characters and occur more than once can

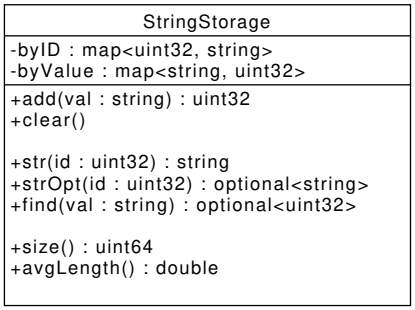


Figure 5.6.: Class diagram of the `StringStorage` class.

actually profit from this approach. Measurements in Chapter 7 show how many entries a dictionary for real-life corpora has.

Using a dictionary is not only useful to save space, but also to make it easier to write data structures that operate on these strings. For example, when implementing containers where labels are stored, only the numeric references to the strings have to be stored in the containers. The references have a constant size which makes it possible to use optimized container implementations and potentially more optimal usage of the main memory cache. Also, testing for equality is much faster since only the numeric IDs need to be compared and there is no need to fetch the actual strings from memory locations that are far away from the data container.

While `StringStorage` is not a C++ Standard Template Library (STL) container by itself, it has simple functions for manipulating the dictionary. The function `add(val:string):uint32` takes a reference to a string as argument and adds it to the dictionary if the string is not already part of it. It returns the newly created ID (the user can not influence which ID a string gets). If the string already exists, the existing ID is returned. With `clear()` it is possible to remove all existing entries from the dictionary. The functions `size():uint64` and `avgLength():double` are useful for statistical purposes. While `size():uint64` only returns the number of elements in constant time, `avgLength():double` does a calculation of the average length of all strings and iterates over all strings in the dictionary.

For retrieving the string value for known IDs, the `str(id:uint32):string` and `strOpt(id:uint32):optional<string>` functions are used. The first one is very efficient because it only returns a reference to the string, but does not copy it. It should only be used if the calling function can ensure that the ID is part of the dictionary because it can not return a reference to a non-existing string and will throw an exception whenever an ID is unknown. The `strOpt(id:uint32):optional<string>` function encodes whether the ID was found in the returned value (using the special `boost:optional` type[14]) but is less efficient than `str(…)` because it needs to copy the string. These functions are usually called when strings are intended to be used as output, or if more complex string comparison than the equal operator is needed. In the case of such a more complex string comparison for two existing labels, it is already

[14]Boost.Optional is documented at `http://www.boost.org/doc/libs/1_64_0/libs/optional/doc/html/` (last accessed 2017-10-25).

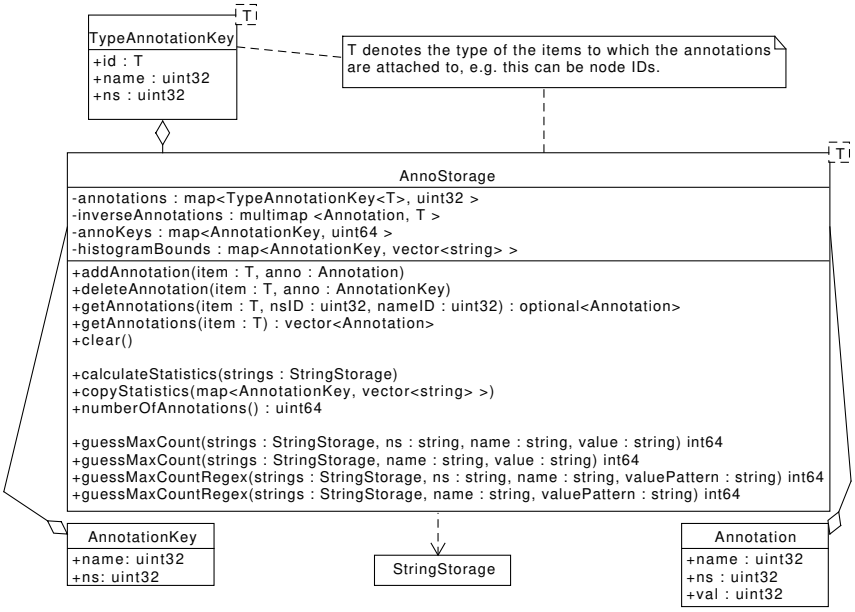


Figure 5.7.: Class diagram of the `AnnoStorage` class and some of the referenced data types.

known that both string IDs have to exist and thus the `str(…)` function can be used, which avoids additional overhead for copying the string. When the string is known, but the ID is needed, the `find(val:string):optional<uint32>` can be used. This function is mainly called during query parsing to get the underlying ID of a given string that is then used in the internal query processing instead of the actual string.

5.4. Annotation storage and search

Any annotation search system must store and handle its basic annotation units, which in case of graphANNIS are modeled as labels on nodes and edges. Labels are attached to different kind of objects but share common properties. The common `AnnoStorage` class is used to represent and organize them. It is parameterized with the type of elements a label can be attached to. For nodes the type parameter are node IDs (`uint32`)[15] and for edges it is a pair of two node IDs. A class diagram of the `AnnoStorage` class is shown in Figure 5.7. Similar to the `StringStorage` class, the most basic members are the map from the annotation key (containing the item, name and namespace ID) to the annotation value (represented as ID) and the inverse map that assigns a set of elements to a fully qualified annotation. New annotations can be added to an element with the `addAnnotation(…)`, and existing ones can be deleted with the `deleteAnnotation(…)` functions.[16] Also, there are getter functions

[15]Node IDs are currently encoded as unsigned 32-bit integers, but graphANNIS can also be configured to use 64-bit integers instead.

[16]These functions are used by the `CorpusStorageManager` class (see Section 5.8) to import existing corpora in the relANNIS format or to apply updates via the external API.

Algorithm 5.1 Equi-depth histogram estimation of maximum number of values matching a range of strings. h is an equi-depth histogram vector, u the universe size (number of all values) and the strings s_{lower}/s_{upper} define the search range.

```
1: function GUESSMAXCOUNT(h, u, s_lower, s_upper)
2:     m ← 0
3:     if |h| ≥ 2 then
4:         for all i ∈ [0, |h| − 2] do
5:             if h_i ≤ s_upper ∧ s_lower ≤ h_{i+1} then ▷ bucket range overlaps search range
6:                 m ← m + 1
7:         return (m / |h|) · u                          ▷ multiply selectivity with universe size
```

(`getAnnotations(…)`) that allow to get the annotation value for a given element given a specific annotation key or a vector of all annotations of an element. There are several extra classes that are used to provide iterators for annotation search, and they can directly access the data structures of the annotation storage. These classes are

- `ExactAnnoKeySearch` for searching for annotations by their (optional) namespace and name,
- `ExactAnnoValueSearch` for searching for annotations by their namespace/name and their exact value,
- `RegexAnnoSearch` for searching for annotations by their namespace/name and values matching a regular expression,
- `NodeByEdgeAnnoSearch` for searching for nodes that have an outgoing edge in a specific edge component and optionally a specific edge annotation.

The `AnnoStorage` class is not only responsible for storing the actual labels, but also for keeping statistics such as the number of all known annotation keys. Together with a histogram of values for each annotation key this statistical information can be used to estimate the number of labels having a certain annotation key and value. This is used by the query planner to estimate (intermediate) result sizes for a query and generated optimized execution plans (see Section 6.2.1). There are several functions with the prefix `guessMaxCount` which implement this estimation. Some of these functions are used to estimate the number of exact matches while others can also estimate the number of matches for a given annotation key and a regular expression for the value. Since updating the statistics is an expensive process, it is necessary to explicitly trigger the creation of new statistics with the `calculateStatistics(…)` function or by copying the statistics of another annotation storage.

The histograms used are equi-depth histograms with random sampling as described in Piatetsky-Shapiro and Connell (1984). Algorithm 5.1 presents the function that returns the estimated number of labels matching a certain value range based on a single equi-depth histogram. GraphANNIS calculates a histogram for the values of all distinct annotation keys. If a search operates on more than one annotation key, the estimates are accumulated. For regular expression searches, the possible match

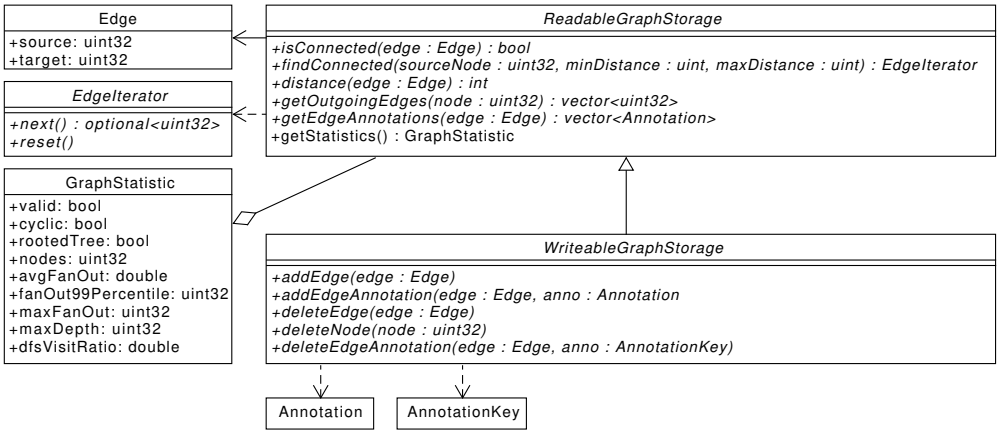


Figure 5.8.: Class diagram of the graph storage base classes and some of the referenced data types.

range for a regular expression pattern is calculated with the help of the RE2 library and used as lower and upper bound. A maximum of 2500 annotation values are sampled per annotation key and the maximum number of buckets is 250. Using equi-depth histograms for estimating the number of labels allows adapting to different non-normal distributions. The computational overhead of using this type of histogram is not problematic, because graphANNIS is an application where corpora are typically imported once and not updated afterwards. Equi-depth histograms where chosen because of their usage in the previous versions of the PostgreSQL database. Note that there are alternative histogram types that might have better estimates for Zipf-like distributions as discussed in Poosala et al. (1996).

5.5. Graph storages

As described in Section 4.1, graphANNIS partitions the data into components. This principle is not only used for distinguishing the different kind of edge annotations in the model, but is also the base for an optimized implementation of the different AQL operators. Each component is identified by the unique combination of its layer name, component name and type. Depending on the annotation that a component represents, their graph structure might be very different. An important design goal of graphANNIS is to make use of these different kinds of graph structures and to allow optimized implementations for them. “Optimized” here means that the data structure should be efficient in their memory usage and that finding reachable nodes is executed as fast as possible for typical annotation types stored in this component.

Figure 5.8 gives an overview of the methods that each graph storage implements. In general, two types of graph storages can be defined:

- an immutable `ReadableGraphStorage` which is instantiated once, and
- a `WriteableGraphStorage` that allows to change the content of the graph storage dynamically.

GraphANNIS typically imports a corpus once and does not change its contents thereafter. Thus, this distinction allows implementing efficient read-only graph storages which use optimizations that might have a large initial overhead at creation time (for example by creating several indexes). Only the `WriteableGraphStorage` defines the methods needed to manipulate edges and edge annotations of a component.

The more general `ReadableGraphStorage` defines methods for accessing the existing edges and edge annotations. The `getOutgoingEdges(node: uint32)` function returns a vector containing all node IDs which are the target of the outgoing edges of the given source node inside the component. Additionally, there are methods that allow efficient reachability queries for a given start node, namely:

- `findConnected(sourceNode : uint32, minDistance : uint, maxDistance : uint) : EdgeIterator`,
- `distance(edge : Edge) : int`, and
- `isConnected(edge : Edge) : bool`.

The implementation of these methods is different for each implemented type of graph storage and is described later in this section. `findConnected(…)` returns an iterator for all edges that are reachable from the start node and have a specified distance. An iterator was chosen to allow lazy evaluation implementations instead of always precalculating all reachable nodes. This is especially useful as their number may be very large and a query might not always need to evaluate all reachable nodes. Giving a range of valid distances as argument to `findConnected(…)` also helps to keep the number of reachable nodes as small as possible and allows more specific implementations. While `findConnected(…)` can be used to generate possible matches of a query, the `distance(…)` function allows filtering existing pairs of nodes. Depending on the specific graph storage, it might be possible to have a more efficient implementation for `distance(…)` instead of just calling `findConnected(…)` with the source node and check if the target node is in the set of returned reachable nodes. If the distance is not needed for filtering, the `isConnected(…)` method can be used instead of `distance(…)`. While its output is not as specific as the other functions, it also allows for optimizations in the implementation.

Another important method of the graph storage is the `getStatistics(…)` function. It returns properties of the structure of the graph that are useful for estimating the result size of joins (see Section 6.2.1 for more details). Since these are expensive to calculate, their generation must be explicitly triggered. Statistics contain, among other, information on:

- whether the sub-graph of the component is cyclic,
- whether it satisfies the tree properties,
- the typical number of outgoing edges per node (average, 99th percentile and maximum fan-out), and
- the length of the longest path.

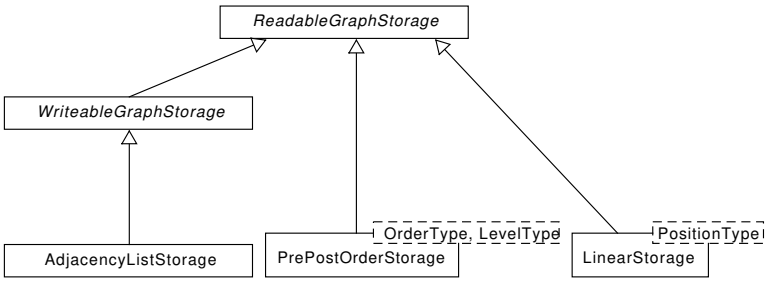


Figure 5.9.: Hierarchy of different GraphStorage implementations.

Currently, three different implementations for graph storages exist (see Figure 5.9 for an overview). They are described in more detail in the next sections. It is possible to add new graph storage implementations to graphANNIS without changing the other parts of the query system. Thus, if new corpora contain annotations with specific edge structures that can not be efficiently queried by any of the existing graph storages, it is possible to add a new graph storage that supports this new type of annotation efficiently instead of re-optimizing the whole system and possible introducing performance regressions for existing corpora.

5.5.1. Adjacency list

The `AdjacencyListStorage` class implements a graph storage by using a unique set of edge entries, sorted by the ID of the source and target node. It is currently the only graph storage that implements the `WriteableGraphStorage` interface and allows to change the content of the graph storage dynamically. Each edge entry in the sorted set is a pair of the ID of the source node and the ID of the target node. Using a set (instead of an array containing lists of outgoing edges) allows checking for uniqueness of an edge with an upper complexity bound of $O(\log n)$, because the set is implemented with a B-tree (Cormen et al. 2009, p. 492). It also allows lookups for all outgoing edges of any node using

- an initial lookup of the first edge of a given node with a complexity of $O(\log n)$,
- a lookup of the successor of the current edge inside the set, which is the potential next edge for the same node, with the same complexity (Cormen et al. 2009, p. 292).

Thus, getting all outgoing edges for a single node in a graph with n nodes and a maximum degree of d has the complexity $O(d \log n)$. In addition to the set of edges the class also contains a set of inverted edges (where the source and target nodes are switched). Currently this is only used to make deletion of nodes more efficient, because they have to be deleted from all edges where there are either the source or the target node. In the future, the inverted edge set would also allow to traverse the graph in inverse direction efficiently (see Section 7.1.2 for a discussion why this useful).

`AdjacencyListStorage` uses graph traversal to implement reachability queries. In particular, the `findConnected(…)` function is implemented with a depth-first

Algorithm 5.2 Implementation of a Depth-First-Search (DFS) for adjacency lists which outputs each found node only once.

```
1:  procedure UNIQUEDFS(node, dist_min, dist_max)
2:      output ← ∅                                          ▷ init with empty set
3:      stack ← emptyStack                                  ▷ init with empty stack

4:      PUSH(stack, (node, 0))                  ▷ add initial node with distance 0 to stack
5:      while r ← NEXT(stack, dist_min, dist_max, output) do
6:          emit r

7:  function NEXT(stack, dist_min, dist_max, output)
8:      found ← false
9:      while found = false ∧ ¬EMPTY(stack) do
10:         (node, dist) ← TOP(stack)
11:         found ← ENTERNODE(node, dist, stack, dist_min, dist_max)
12:     if found then
13:         return (node, dist)
14:     else
15:         return false

16: function ENTERNODE(node, dist, stack, dist_min, dist_max, output)
17:     found ← false
18:     POP(stack)
19:     if dist ≥ dist_min ∧ dist ≤ dist_max then      ▷ check if distance in valid range
20:         found ← true
21:     if dist ≤ dist_max then                        ▷ add all child nodes to the stack
22:         for all child ∈ OUTGOINGEDGES(node) do
23:             PUSH(stack, (child, dist + 1))
24:     if found then
25:         if node ∉ output then                      ▷ only output node once
26:             output ← output ∪ {node}
27:             return true
28:     return false
```

function	time complexity
`findConnected(…)`	$O\left(\lvert V_r\rvert + (\lvert E_r\rvert \cdot log\lvert E\rvert)\right)$
`distance(…)`	$O\left(\lvert V_r\rvert + (\lvert E_r\rvert \cdot log\lvert E\rvert)\right)$
`isConnected(…)`	$O\left(\lvert V_r\rvert + (\lvert E_r\rvert \cdot log\lvert E\rvert)\right)$

Table 5.1.: Complexity for the reachability functions of `AdjacencyListStorage`. $|V_r|$ is the number of nodes reachable by the source node, $|E_r|$ the number of all edges that belong to a path that connects the source node with a reachable node. $|E|$ is the number of all edges in the graph.

traversal that additionally checks if each node is used as output only once, which adds a computational overhead.[17] See Algorithm 5.2 for a listing of the pseudo-code. The traversal will stop as soon as the maximum allowed distance is reached. If a node is reachable via more than one path with valid lengths, it will be still outputted only once (Algorithm 5.2, line 25). Testing for cycles is not performed for the `findConnected(…)` function to avoid the additional overhead introduced by testing each traversed node and since most components are cycle-free anyway. The traversal will still always stop, because a finite maximum distance must be given as argument. Currently, only the `LEFT_TOKEN` and `RIGHT_TOKEN` components contain cycles (see Section 4.2.3 for a description) and these components are typically used by AQL operators that query them with a maximum distance of 1.

The `distance(…)` function also uses a depth-first traversal, but stops exploring a node if it is part of a cycle. Its pseudo-code is given in Algorithm 5.3. The same cycle-safe depth-first traversal implementation is used in the `isConnected(…)` function. Initializing the traversal has a constant computational overhead. For the special (but frequent) case that the requested distance for `isConnected(…)` is exactly one, a simplified implementation that just checks the existence of the specific edge in the edge set is used in order to avoid this overhead.

In general, the running time of a depth-first search is $O(|V| + |E|)$, where $|V|$ is the number of nodes and $|E|$ is the number of edges (Cormen et al. 2009, p. 606). Each node can be visited only once, which adds a linear complexity relative to the number of nodes to the overall complexity. For each visited node, all outgoing edges are retrieved in a loop, but each edge will also be visited only once in total. An ideal adjacency list needs constant time to lookup a list of all outgoing edges of a single node, however, `AdjacencyListStorage` needs logarithmic time for this task. Also, not all nodes and edges are actually searched, but only the reachable ones. The worst-case running times for the reachability functions of `AdjacencyListStorage` are given in Table 5.1 and reflect these considerations. While the worst-case times are the same, the cycle-safe implementation needs an additional check if a searched node is part of the current path to detect cycles. The UniqueDFS procedure in Algorithm 5.2 also checks for unique results, but this check is only executed if a node is part

[17] Currently, a red-black tree (Cormen et al. 2009, pp. 308 ff.) based set from the default C++ STL is used to store the unique result. Thus, an additional lookup cost of $O(\log_2 n)$ is needed for each node that fulfills the path length constraint.

Algorithm 5.3 Implementation of a cycle-safe DFS for adjacency lists.

```
1: procedure cycleSafeDFS(node, dist_min, dist_max)
2:     dist_last ← 0
3:     path ← emptyList                                  ▷ init with empty list
4:     stack ← emptyStack                                ▷ init with empty stack

5:     push(stack, (node, 0))                ▷ add initial node with distance 0 to stack
6:     while r ← next(stack, dist_min, dist_max, dist_last, path) do
7:         emit r

8: function next(stack, dist_min, dist_max, dist_last, path)
9:     found ← false
10:    while found = false ∧ ¬empty(stack) do
11:        (node, dist) ← top(stack)
12:        found ← enterNode(node, dist, stack, dist_min, dist_max, dist_last, path)
13:    if found then
14:        return (node, dist)
15:    else
16:        return false

17: function enterNode(node, dist, stack, dist_min, dist_max, dist_last, path)
18:    if dist_last ≥ dist then                ▷ test if subgraph was completed
19:        removeIdx(path, [dist, length(path)])      ▷ remove all below the parent
20:    if contains(path, node) then                                ▷ test for cycle
21:        dist_last ← dist
22:        return false
23:    else
24:        insert(path, node)
25:        dist_last = dist
26:        found ← false
27:        pop(stack)
28:        if dist ≥ dist_min ∧ dist ≤ dist_max then  ▷ check if distance in valid range
29:            found ← true
30:        if dist ≤ dist_max then                  ▷ add all child nodes to the stack
31:            for all child ∈ outgoingEdges(node) do
32:                push(stack, (child, dist + 1))
33:        return found
```

of the output. This can give a slight performance advantage for queries that have a minimum distance greater than one, because the number of output nodes is smaller than the number of reachable nodes.

5.5.2. Pre-/post-order encoding

The original relANNIS implementation relied on the pre-/post-order encoding to speed up the search for reachable nodes (see Section 3.3.1). Pre-/post-order encoding can be used on all Directed Acyclic Graphs (DAGs), but there are multiple order-entries for nodes that are reachable by more than one path (Rosenfeld 2010). Thus, this encoding works best for graphs that are (almost) trees. Since many linguistic annotations have this property, a specialized graph storage using pre-/post-order encoding is very useful for graphANNIS and was implemented as immutable graph storage.

In pre-/post-order encoding, each node gets one or more order entries assigned. Each entry consists of three fields: the pre- and post-order values and an additional level value. The legacy relANNIS implementation always uses the same data type `integer` for this field, which is a 32 bit signed integer type in PostgreSQL[18]. This is due to the nature of the fixed database table layout, where a column must be valid for all possible row values. In contrast, because graphANNIS allows defining different kinds of graph storages for different components, the data type for the fields can be optimized. Thus, the `PrePostOrderStorage` class is parametrized with the types `OrderType` and `LevelType`. This first data type is used for the pre- and post-order fields and the second one is used for the level field. Since this is an immutable graph storage where the data is not changed after the initial import, the optimal data types can be chosen to depend on the number of edges and the maximum depth of the paths of a component. An optimization of the data types allows for more efficient memory usage but could also enhance the processing speed by better cache usage or faster CPU instructions for smaller data types.

Since the `PrePostOrderStorage` is immutable, computation of the pre-/post-order values has to be done only once after the initial import. In order to compute the index, the root nodes of the original graph are iterated using the cycle-safe depth-first iterator presented in Section 5.5.1. If a cycle is detected during traversal, the index creation is aborted and an error is emitted. Similar to the `AdjacencyListStorage`, the resulting order entries are stored in two maps: one multi-map which holds all order entries for a node ID and an inverse map which maps each order entry to its node. The complete pseudo-code for the generation of the pre-/post-order is given in Algorithm 5.4.

The two index maps that are created initially are used to allow an efficient implementation of the reachability functions (see Equation 3.1). Algorithm 5.5 describes the implementation of the `findConnected(…)` function. The implementation first finds all order entries for the given start node (line 3). In line 4, all order values with a pre-order value larger than the one for the start node are searched in the `order2node` map and it is checked whether the other conditions are fulfilled, too. Since the `order2node`

[18] `https://www.postgresql.org/docs/9.6/static/datatype-numeric.html` (last accessed 2017-10-25)

Algorithm 5.4 Calculation of the pre- and post-order which iterates over the nodes using the cycle safe DFS from Algorithm 5.3.

```
1: procedure calculatePrePost
2:     node2order ← emptyMultiMap
3:     order2node ← emptyMap
4:     order ← 0
5:     for all r ∈ rootnodes do
6:         dist_last ← 0
7:         stack ← empty

8:         enterNode(r, 0)                    ▷ add root with order and level 0 to stack
9:         for all (node, dist) ∈ cycleSafeDFS(r, 1, ∞) do          ▷ DFS-ordered
   iteration
10:            if dist > dist_last then                              ▷ check if first visit
11:                enterNode(node, dist)
12:            else   ▷ is neighbor node, post-order can be assigned for stack entries
13:                while length(stack) > dist do
14:                    exitNode
15:                enterNode(node, dist)                  ▷ put new node on the stack
16:            dist_last ← dist
17:        while ¬empty(stack) do          ▷ process remaining entries from stack
18:            exitNode

19: procedure enterNode(node, level)
20:     pre ← order
21:     order ← order + 1
22:     push(stack, (node, pre, level))   ▷ add node with pre-order and level to stack

23: procedure exitNode
24:     (n, pre, level) ← top(stack)     ▷ get pre-order and level for node from stack
25:     post ← order                                  ▷ post-order is the current order
26:     order ← order + 1
27:     pop(stack)

28:     node2order[n] ← (pre, post, level)                ▷ add order tuple to node
29:     order2node[(pre, post, level)] ← n    ▷ assign distinct node ID to order tuple
```

Algorithm 5.5 Pseudo-code for implementation of `findConnected(…)` for the pre-/post-order based graph storage.

```
1: procedure findConnectedPrePost(node, dist_min, dist_max)
2:     visited ← ∅                                              ▷ init with empty set
3:     for all (pre_start, post_start, level_start) ∈ node2order[node] do
4:         for all (node, (pre, post, level)) ∈ order2node : pre ≥ pre_start do   ▷ use pre_start as initializer for iterator
5:             d_level ← |level − level_start|
6:             if post ≤ post_start ∧ dist_min ≤ d_level ≤ dist_max then   ▷ check reachability
7:                 if node ∉ visited then                       ▷ output only once
8:                     visited ← visited ∪ {node}
9:                     emit node
10:            else if pre ≥ post_start then
11:                break                                        ▷ abort inner loop
```

function	time complexity
`findConnected(…)`	$O\left(f^2 \cdot \lvert V_o\rvert \cdot \log^2\left(f \cdot \lvert V\rvert\right)\right)$
`distance(…)`	$O\left(f^2 \cdot \log^2 \lvert V\rvert\right)$
`isConnected(…)`	$O\left(f^2 \cdot \log^2 \lvert V\rvert\right)$

Table 5.2.: Complexity for the reachability functions of `PrePostOrderStorage`. $|V_o|$ is the number of output nodes, $|V|$ the number of all nodes and f is the duplication factor of the order entries ($f = 1$ means no duplication).

map is implemented using a B-tree with the pre-order value as its key, the lookup of each pre-order value has an upper complexity bound of $O(\log n)$ (Cormen et al. 2009, p. 492). Each node is only emitted once, which needs an additional check (line 7). The inner loop is aborted when the pre-order value from the candidate node is larger than the post-order value from the start node (line 11). While `findConnected(…)` needs to iterate over a range of pre-orders, both the `distance(…)` and `isConnected(…)` functions can query the order entries for both given nodes directly and then compare the queried entries if they fulfill the reachability criteria. See Algorithm 5.6 and 5.7 for details.

The running time for `findConnected(…)` is sensitive to the output size, in this case the number of output nodes $|V_o|$. Algorithm 5.5 has two loops: The outer one finds all order entries for the given start node and the inner one finds all matching order values for all output nodes. Assuming there are at most f order values per node, $|V|$ is the number of all nodes and that the lookup of map entries has complexity $O(\log n)$, the running time of the two nested loops can be expressed as:

$$O\left((f \cdot \log|V|) \cdot (f \cdot |V_o| \cdot \log(f \cdot |V|))\right) \tag{5.3}$$

$$= O\left(f^2 \cdot |V_o| \cdot \log|V| \cdot \log(f \cdot |V|)\right) \tag{5.4}$$

$$= O\left(f^2 \cdot |V_o| \cdot \log^2(f \cdot |V|)\right) \tag{5.5}$$

Algorithm 5.6 Pseudo-code for implementation of `distance(…)` for the pre-/post-order based graph storage.

```
1: function distancePrePost(source, target)
2:     if source = target then
3:         return 0
4:     else
5:         found ← false
6:         d_min ← ∞
7:         for all (pre_source, post_source, level_source) ∈ node2order[source] do
8:             for all (pre_target, post_target, level_target) ∈ node2order[target] do
9:                 if pre_source ≤ pre_target ∧ post_target ≤ post_source then
10:                    d_level ← level_target − level_source
11:                    if d_level ≥ 0 then
12:                        found ← true
13:                        d_min ← min(d_min, d_level)
14:        if found then
15:            return d_min
16:        else
17:            return −1
```

Algorithm 5.7 Pseudo-code for implementation of `isConnected(…)` for the pre-/post-order based graph storage.

```
1: function isConnectedPrePost(source, target, dist_min, dist_max)
2:     for all (pre_source, post_source, level_source) ∈ node2order[source] do
3:         for all (pre_target, post_target, level_target) ∈ node2order[target] do
4:             if pre_source ≤ pre_target ∧ post_target ≤ post_source then
5:                 d_level ← |level_target − level_source|
6:                 if dist_min ≤ d_level ≤ dist_max then
7:                     return true
8:     return false
```

Compared to the running time of `AdjacencyListStorage`, that one of `PrePostOrderStorage` does not depend on the number of reachable nodes, but on the ones that are part of the output. Also, the number of edges is only indirectly part of the running time complexity via the duplication factor f. For graph storages that are a tree and thus $f \leq 1$, the complexity is reduced to $O\left(|V_o| \cdot \log^2(|V|)\right)$. Table 5.2 lists the running time complexities for the other reachability functions `distance(…)` and `isConnected(…)`. These depend on the duplication factor, too. But they are only applied to a single output candidate and thus $|V_o|$ is eliminated. This is an improvement compared to `AdjacencyListStorage`, where running time of all three reachability functions depend on the number of reachable nodes.

5.5.3. Linear graphs

Pre-/post-order encoding works well for trees, but it requires three values to encode the position of a node inside the tree hierarchy. For linear graphs, this is inefficient. Linear graphs are trees with a maximum out degree of 1. Thus, they consist of a set of disjoint paths, where each path consists of a sequence $v = v_1, v_2, \ldots, v_n$ of nodes. This allows the assignment of the index of the node inside the path as single and distinctive order value to each node. Also, there are no duplicate order entries for each node, which simplifies the implementation of the reachability functions. The `LinearStorage` class is also parametrized with the type of the order entry value. This allows the optimization of the space usage. Depending on how many nodes are part of a component, a smaller or larger data type can be used.

Algorithm 5.8 Pseudo-code for implementation of `findConnected(…)` for the linear graph based graph storage.

```
1: procedure findConnectedLinear(node, dist_min, dist_max)
2:     if node2pos[node] ≠ ∅ then
3:         (root, pos) ← node2pos[node]
4:         v ← nodeChains[root]
5:         for all i ∈ [(pos + dist_min), min(|v|, pos + dist_max)] do
6:             emit v_i
```

Algorithm 5.9 Pseudo-code for implementation of `distance(…)` for the linear graph based graph storage.

```
1: function distanceLinear(source, target)
2:     if node2pos[source] ≠ ∅ ∧ node2pos[target] ≠ ∅ then
3:         (root_source, pos_source) ← node2pos[source]
4:         (root_target, pos_target) ← node2pos[target]
5:         if root_source = root_target ∧ pos_source ≤ pos_target then
6:             return true
7:     return false
```

Algorithm 5.10 Pseudo-code for implementation of `isConnected(…)` for the linear graph based graph storage.

```
1: function isConnectedLinear(source, target, dist_min, dist_max)
2:     if node2pos[source] ≠ ∅ ∧ node2pos[target] ≠ ∅ then
3:         (root_source, pos_source) ← node2pos[source]
4:         (root_target, pos_target) ← node2pos[target]
5:         if root_source = root_target ∧ pos_source ≤ pos_target then
6:             if dist_min ≤ |pos_source − pos_target| ≤ dist_max then
7:                 return true
8:     return false
```

function	time complexity
`findConnected(…)`	$O(\|V_o\| + \log\|V\|)$
`distance(…)`	$O(\log\|V\|)$
`isConnected(…)`	$O(\log\|V\|)$

Table 5.3.: Complexity for the reachability functions of `LinearStorage`. $|V_o|$ is the number of output nodes and $|V|$ the number of all nodes.

In order to store the set of paths, the `LinearStorage` class uses two maps. One maps each node ID to its relative position and is called `node2pos`. A relative position is defined by the ID of the root node of a path and the index of the position of the node inside this path. The other map, called `nodeChains` stores the actual node sequence for each root node of the component as a vector. These two maps allow to efficiently compare two given nodes (which is needed by the `isConnected(…)` and `distance(…)` functions), as well as finding reachable nodes for a given start node (as needed by the `findConnected(…)` function). The implementation of these functions is described in Algorithm 5.8, 5.9 and 5.10. As each node can only have one order entry assigned per component, no uniqueness checks are needed for the `findConnected(…)` function.

Table 5.3 lists the running time complexity functions for the reachability functions of the `LinearStorage`. The `findConnected(…)` function is sensitive to the output size $|V_o|$, because all found nodes need to be emitted. A single lookup to a map with complexity $O(\log n)$ is needed to find the vector of nodes for a given start node (line 3 of Algorithm 5.8). Together with the loop over this vector, the overall complexity is $O(|V_o| + \log|V|)$. Both `distance(…)` and `isConnected(…)` do not need to iterate over a result and thus only have a running time complexity of $O(\log|V|)$.

5.6. Operators

The classes that have been described above are suitable for different kinds of labeled graphs and not specific to AQL. This section describes implementations of AQL operators. Joins make use of these operator implementations: The join is responsible for combining the tuples of a LHS and RHS, but the operators map the logical definition of an AQL operator to an actual implementation. Separating the implementation of the joins and the operators, and pairing them in an `ExecutionNode` allows for a flexible combination of join types and operators. The operator implementations are responsible for

- fetching candidate triples for a RHS that fulfill the logical operator definition for a given LHS, or
- check if a given pair of triples fulfills the logical operator definition.

In order to implement this functionality, they need access to the database. Each operator implementation uses the general graph storage functions to implement its specific semantics and has access to all components of the graph. It does not need to

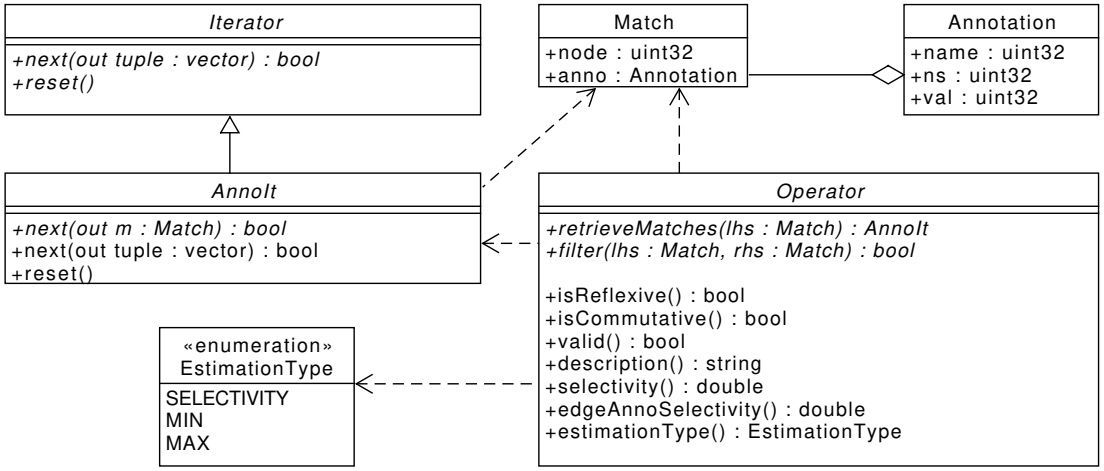


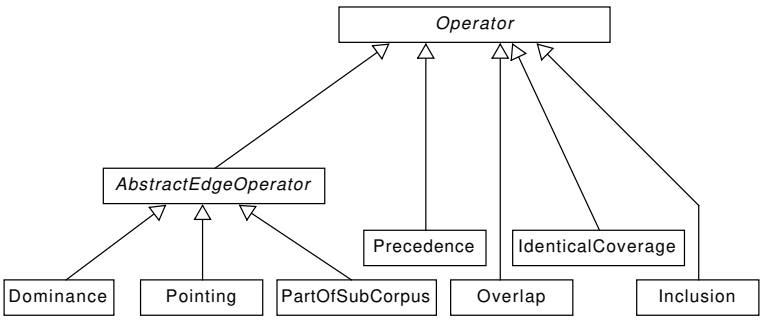
Figure 5.10.: Class diagram of the `Operator`, `Iterator`, `AnnoIt` and related classes.

know which implementation is used for each graph storage, but can assume that the most efficient implementation is chosen for a specific component.

Figure 5.10 shows the diagram of the `Operator` class and the signature for the functions any operator must implement. The most important ones are the `retrieveMatches(lhs: Match):AnnoIt` and the `filter(lhs: Match, rhs: Match):bool` functions. This interface is similar to the `ReadableGraphStorage` interface with its `findConnected(…)` and `isConnected(…)` functions, but an operator does not operate on nodes, but on match triples (which include a node ID and the qualified label name). Operators can operate on more than one graph storage and can express more complex logic than just reachability on a single edge component. The `retrieveMatches(…)` function takes a LHS match as argument and produces an iterator over all matching RHSs. If match candidates for both the LHS and RHS already exist, the `filter(…)` function, which returns *true* or $false$, can be used as predicate.

In addition to these basic functions, all operators need to implement several functions needed for query optimization and debugging. If the `isReflexive()` function returns *true*, an operator implementation indicates that a single match can have the same node as both LHS and RHS. This information is needed by join operations in order to emit the correct results. The `isCommutative()` function is used to indicate if both arguments of the operator can be exchanged without changing the result, which is very helpful for optimizing queries. In some cases, an operator already knows at construction time that it can not produce any results, for example because a required edge component does not exist or is empty. In this case the `valid()` function will return $false$ and allows the query plan to statically return an empty result. The `selectivity()`, `edgeAnnoSelectivity()` and `estimationType()` functions are used to estimate the result size of a join between two execution nodes, when the join implementation uses this specific operator. When visualizing the query plans, the `description()` function is used to get a textual representation of the operator.

In the next section, the operators implemented by graphANNIS are described in more detail. An overview of which operators exist, how they relate to each other and which



	reflexive	commutative
`Dominance`	✓	✗
`Pointing`	✓	✗
`PartOfSubCorpus`	✗	✗
`Precedence`	✓	✗
`Overlap`	✗	✓
`IdenticalCoverage`	✗	✓
`Inclusion`	✗	✗

Figure 5.11.: Hierarchy of different `Operator` implementations and table with their logical properties. "✓" means the operator has the property, "✗" that it does not.

retrieveMatches(…)/**f**ilter(…)	`>` r	`>` f	`->` r	`->` f	`.` r	`.` f	`_=_` r	`_=_` f	`_i_` r	`_i_` f	`_o_` r	`_o_` f	`meta::` r	`meta::` f
`DOMINANCE`	✓	✓												
`POINTING`			✓	✓										
`ORDERING`					✓	✓			✓	✓		✓		
`LEFT_TOKEN`					✓	✓	✓	✓	✓	✓		✓		
`RIGHT_TOKEN`					✓	✓	✓	✓	✓	✓		✓		
`COVERAGE`											✓			
`INVERSE_COVERAGE`											✓			
`PART_OF_SUBCORPUS`													✓	✓

Table 5.4.: Matrix of graph storage types used by the different operator functions.

logical properties they have is given in Figure 5.11. Each operator implementation of `retrieveMatches(lhs: Match):AnnoIt` and `filter(lhs: Match, rhs: Match) :bool` needs different types of graph storages as argument, and Table 5.4 gives an overview which implementation uses which graph storage type.

5.6.1. Dominance and pointing relation operators

The dominance operator (`>`) and the pointing relation operator (`->`) of AQL are very similar, because they are both defined to match nodes that are connected by a path inside a component with optional additional restrictions on the edge annotations of this path. The difference is, that for the dominance operator

1. multiple components can be searched because the name of the component is optional, and

2. it is required that all components of the type `DOMINANCE` taken together are cycle-free.[19]

In contrast, the pointing relation operator always operates on a single component and there is no restriction on cycles for a combination of components. Both operators are implemented by using the same abstract base class `AbstractEdgeOperator`. This class takes a set of components and an optional edge annotation constraint as argument and implements the `retrieveMatches(…)` function by iterating over all graph storages, returning the result of their `findConnected(…)` function with an additional check on the edge constraint if necessary. If more than one graph storage is given as argument, an internal set of results is used to make sure only unique results are returned. The `filter(…)` function is implemented similarly, by iterating over the set of all given components and returning *true* if the `isConnected(…)` is *true* for any of the components, and if the edge annotation constraint is fulfilled.

Selectivity for operators that inherit `AbstractEdgeOperator` is calculated by calculating the selectivity for each graph storage in the given set and using the most pessimistic one as selectivity for the complete operator. The basic idea is to estimate the number of reachable nodes for a single source node for both the maximum and minimum path length and dividing the difference by the number of nodes in total. Estimating the number of reachable nodes is performed by assuming that the graph component is a complete k-ary tree (Cormen et al. 2009, p. 1179) and the average number of outgoing edges (`avgFanOut`) from the graph statistics is the degree k. Compared to an actual k-ary tree, the average fan-out can be a fractional number and also smaller than 1. In general, the number of nodes in a complete k-ary tree with height h is $\frac{k^h-1}{k-1}$ (Cormen et al. 2009, p. 1179). For $k \leq 1$ this formula is not valid and a simplified estimation is used instead. Given an average fan-out of k, a path length constraint (d_{min}, d_{max}) denoting the minimal and maximal path length

[19] The dominance operator is typically used to represent syntactic annotations. These divide for example a sentence into hierarchical structures. If such a hierarchical node "dominates" another one, this also implies it inherits the text coverage from all child nodes. Thus, even if there are several components to represent dominance, the combination of all dominance components must also be cycle-free because a token can not cover itself.

Algorithm 5.11 `AbstractEdgeOperator` selectivity estimation function with a given set GS of graph storages and a restriction on the path length $(dist_{min}, dist_{max})$. as argument.

1: **function** SELECTIVITY($GS, dist_{min}, dist_{max}$)
2: **if** $|GS| == 0$ **then**
3: **return** 0.0 ▷ if graph storage set is empty there is nothing to find
4: $worst \leftarrow 0.0$
5: $n_{total} =$ NUMEROFTOTALNODES() ▷ estimate total number of nodes in DB
6: **for** $g \in GS$ **do** ▷ calculate selectivity for each graph storage
7: $stat \leftarrow$ GETSTATISTICS(g)
8: $l_{min} \leftarrow max(0, dist_{min} - 1)$ ▷ limit minimal query path length
9: $l_{max} \leftarrow min(dist_{max}, stat.maxDepth)$ ▷ limit maximal query path length
10: $k \leftarrow stat.avgFanOut$
11: **if** $k > 1$ **then**
12: $n_{reachable} \leftarrow \lceil \frac{k^{l_{max}}-1}{k-1} \rceil - \lceil \frac{k^{l_{min}}-1}{k-1} \rceil$
13: **else**
14: $n_{reachable} \leftarrow \lceil k \cdot l_{max} \rceil - \lceil k \cdot l_{min} \rceil$
15: $sel \leftarrow \frac{n_{reachable}}{n_{total}}$ ▷ estimate selectivity for this graph storage
16: $worst \leftarrow max(worst, sel)$ ▷ use worst selectivity as overall result
17: **return** $worst$

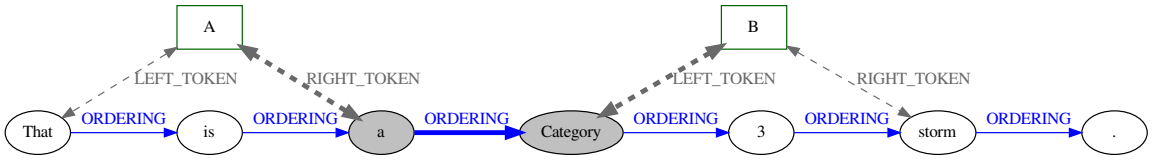


Figure 5.12.: Precedence operator example. The token "a" is precedent to the token "Category" because they are connected by an edge of the `ORDERING` component. Likewise, the span node A precedes B because the tokens that are connected by the `RIGHT_TOKEN` respectively `LEFT_TOKEN` edges are precedent to each other.

and the total number of nodes n_{total}, the selectivity for the `AbstractEdgeOperator` for a single component is defined as:

$$selectivity_{EdgeOp}(k, d_{min}, d_{max}, n_{total}) = \begin{cases} \frac{\lceil \frac{k^{d_{max}}-1}{k-1} \rceil - \lceil \frac{k^{(d_{min}-1)}-1}{k-1} \rceil}{n_{total}} & \text{if } k > 1 \\ \frac{\lceil k \cdot d_{max} \rceil - \lceil k \cdot (d_{min}-1) \rceil}{n_{total}} & \text{if } k \leq 1 \end{cases} \tag{5.6}$$

The pseudo-code of the selectivity estimation is given in Algorithm 5.11. The actual implementation uses the graph based statistics to improve the estimation. For example, instead of using the possible unlimited d_{max} argument that is provided, the calculation clips d_{max} with the measured maximal path length of the graph storage component.

5.6.2. Precedence

Given two tokens, the `Precedence` operator (`.`) checks that they are connected by a path of edges inside an `ORDERING` component with a given length. Non-token nodes inherit their precedence position from the right-most covered token (if it is on the LHS of the operator) or the left-most-covered token (if it is on the RHS of the operator). Thus, the precedence operator implementation needs a specific `ORDERING` component (either the general one with the empty name or a named one for a specific segmentation) and the `LEFT_TOKEN`/`RIGHT_TOKEN` component graph storages as input. An example is given in Figure 5.12 and the implementation of the `filter(…)` and `retrieveMatches(…)` functions is given in Algorithm 5.12 and 5.13.

Selectivity is calculated similarly to the `AbstractEdgeOperator` with the estimated number of reachable nodes per source node divided by the total number of nodes:

$$selectivity_{Precedence}(d_{min}, d_{max}, n_{total}) = \frac{2 \cdot (d_{max} - d_{min} + 1)}{n_{total}} \tag{5.7}$$

Precedence is defined over the ordering of tokens, as it was modeled in Section 4.2.2. Edges of the `ORDERING` component connect tokens that are precedent to each other, pairwise. Conflicting tokenization will result in an additional, named, `ORDERING`

Algorithm 5.12 `Precedence` operator implementation of the `filter(…)` function. It needs the corresponding `ORDERING` (g_O), the `LEFT_TOKEN` (g_L) and the `RIGHT_TOKEN` component (g_R) as arguments. Also, the LHS and RHS node candidates (n_{lhs}, n_{rhs}) and the path length restriction ($dist_{min}, dist_{max}$) are given.

```
1: function FILTER(n_lhs, n_rhs, dist_min, dist_max, g_O, g_L, g_R)
2:     if COMPONENTNAME(g_O) ≠ ∅ then        ▷ named ORDERING components are segmentations, which are only defined on tokens directly
3:         n_start = n_lhs
4:         n_end = n_rhs
5:     else
6:         n_start = CONNECTEDTOKEN(n_lhs, g_R)
7:         n_end = CONNECTEDTOKEN(n_rhs, g_L)
8:     return ISCONNECTED(g_O, n_start, n_end, dist_min, dist_max)

9: function CONNECTEDTOKEN(n, g) ▷ find connected token for a node in a graph storage g
10:    if ISTOKEN(n) then
11:        return n                          ▷ return the node itself because it is a token
12:    else
13:        return GETOUTGOINGEDGES(n, g)[0]  ▷ return first outgoing edge of given component
```

Algorithm 5.13 `Precedence` operator implementation of the `retrieveMatches(…)` function. It needs the corresponding `ORDERING` (g_O), the `LEFT_TOKEN` (g_L) and the `RIGHT_TOKEN` component (g_R) as arguments. Also, the LHS node candidate (n_{lhs}) and the path length restriction ($dist_{min}, dist_{max}$) are given.

```
1: procedure RETRIEVEMATCHES(n_lhs, dist_min, dist_max, g_O, g_L, g_R)
2:     if COMPONENTNAME(g_O) ≠ ∅ then        ▷ named ORDERING components are segmentations, which are only defined on tokens directly
3:         n_start ← n_lhs
4:     else
5:         n_start ← CONNECTEDTOKEN(n_lhs, g_R)
6:     for t ∈ FINDCONNECTED(n_start, dist_min, dist_max, g_O) do   ▷ find connected tokens
7:         for n ∈ GETOUTGOINGEDGES(t, g_L) do
8:             emit n                        ▷ return all nodes left-aligned with the token
9:         emit t                            ▷ also return the token itself
```

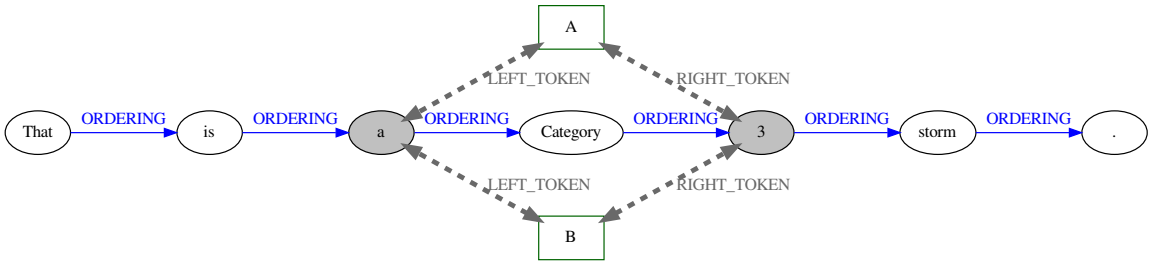


Figure 5.13.: Same text (also called identical coverage) operator example. The span node A has the same text coverage as B because the tokens that are connected by the `RIGHT_TOKEN` respectively `LEFT_TOKEN` edges are the same.

component, but each component itself is always a linear graph.[20] As a consequence, average fan-out estimation is not included in the estimation and the queried path length $d_{max} - d_{min}$ is used directly as number of reachable tokens. As with the `AbstractEdgeOperator` implementation, the maximum distance is clipped to the maximum path length in the component. The estimation $d_{max} - d_{min}$ only returns the number of tokens, but does not include other nodes that cover these tokens. If the source node is a span, the right-aligned token of the span is also not included in this estimation. There is no separate statistics for this kind of information yet and thus the number of reachable nodes is in all cases simply extended by one (to account for the aligned token) and then multiplied by two (to account for other spans).

5.6.3. Same text operator

The same text operator (`_=_`), sometimes also called "same coverage" or "identical coverage", finds nodes that cover exactly the same tokens (which corresponds to covering the same text) and is implemented using the `IdenticalCoverage` class. In relANNIS the operator is implemented by comparing the left and right token index columns in the database. GraphANNIS could use the edges of the coverage components and find nodes that share the same set of covered tokens, but this is inefficient for nodes that cover many tokens (for example spans that mark chapters of a book and cover all tokens of a chapter). Instead, the `LEFT_TOKEN` and `RIGHT_TOKEN` components are used to find the left-most and right-most covered token for both nodes and then the tokens of the LHS and RHS are compared (see Figure 5.13 for an example). The pseudo-code of the implementation is given in Algorithm 5.14 and 5.15.

Selectivity estimation is difficult for this operator. We could try to estimate the probability that two nodes have the same left-most and right-most covered token by estimating these probabilities separately. For example, if each token is assigned an index and for each node the index of the left-most and right-most covered token is

[20] An `ORDERING` component which is not a linear graph could be created in theory. However, since all data is originated from corpora that are either encoded in Salt or the relANNIS format, where this is not allowed, this case will not occur in practice. If future corpora need a more complex concept of token precedence, this should be reflected in the selectivity estimation.

Algorithm 5.14 `IdenticalCoverage` operator implementation of the `filter(…)` function. It needs the corresponding `LEFT_TOKEN` (g_L) and the `RIGHT_TOKEN` (g_R) component as arguments. Also, the LHS and RHS node candidates (n_{lhs}, n_{rhs}) are given.

```
1: function FILTER(n_lhs, n_rhs, g_L, g_R)
2:     start_lhs ← CONNECTEDTOKEN(n_lhs, g_R)
3:     end_lhs ← CONNECTEDTOKEN(n_lhs, g_L)
4:     start_rhs ← CONNECTEDTOKEN(n_lrs, g_R)
5:     end_rhs ← CONNECTEDTOKEN(n_rhs, g_L)

6:     return start_lhs = start_rhs ∧ end_lhs = end_rhs
```

Algorithm 5.15 `IdenticalCoverage` operator implementation of the `retrieveMatches(…)` function. It needs the corresponding `LEFT_TOKEN` (g_L) and the `RIGHT_TOKEN` (g_R) component as arguments. Also, the LHS source node candidate n_{lhs} is given.

```
1: procedure RETRIEVEMATCHES(n_lhs, g_L, g_R)
2:     n_left ← CONNECTEDTOKEN(n_lhs, g_L) ▷ get left-most covered token for source
3:     n_right ← CONNECTEDTOKEN(n_lhs, g_R)      ▷ get right-most covered token for
       source

4:     if n_left = n_right then                  ▷ covered range is exactly one token
5:         emit n_left                                          ▷ output covered token

6:     for n ∈ GETOUTGOINGEDGES(n_left, g_L) do      ▷ find left-aligned non-token
7:         if CONNECTEDTOKEN(n, g_R)[0] = n_right then ▷ check if also right-aligned
8:             emit n
```

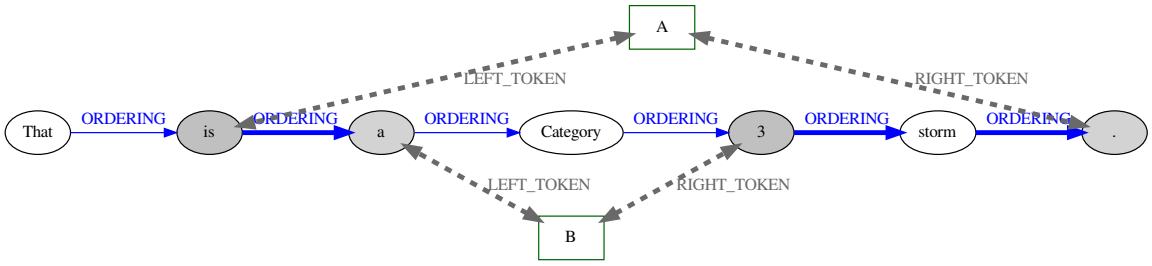


Figure 5.14.: Inclusion operator example. The span node A includes B because there is a path in the `ORDERING` component from the left-most token of A ("is") to the left-most token of B ("a") and one from the right-most token of B ("3") to the right-most token of A (".").

known, the probability that two nodes have the same index value for one of them can be calculated from the sum of all tokens. But these probabilities are not independent of each other (the right-aligned token is always right of the left-aligned one) and cannot simply be multiplied. Using the probability of two nodes having the same number of covered tokens (which corresponds to the length of the span) as second independent probability (in addition to having the same left-most covered token) would be more helpful and could be estimated by using histograms. Because these statistics do not exist yet, a much more simplified assumption is made, namely that there exists one other node with the same text coverage for each node:

$$selectivity_{IdenticalCoverage}(n_{total}) = \frac{1}{n_{total}} \tag{5.8}$$

5.6.4. Inclusion operator

The `Inclusion` operator (`_i_`) filters nodes where the set of covered tokens of the RHS is a subset of the covered tokens of the LHS. As with the precedence operator, calculating and comparing the sets of covered tokens would be too expensive, especially for nodes that cover numerous tokens. Instead, the operator is implemented as path queries between the left-most and right-most token of both nodes. A node x includes another node y if there is both a path from the left-most covered token of x to the left-most covered token of y and one from the right-most covered token of y to the right-most covered token of x. See Figure 5.14 for an example. The length of the paths can be zero if the two nodes are aligned with the same tokens. Algorithms 5.16 and 5.17 show how inclusion is expressed as path queries on the `ORDERING` component.

Calculating the selectivity for the inclusion operator is performed with the help of a per-node fan-out estimation. In contrast to the `AbstractEdgeOperator` we do not use the average fan-out but the 99th percentile value. This is a more pessimistic estimation and was chosen after initial experiments with the prototype on several test corpora. Given f_{cov} as 99th percentile fan-out for the `COVERAGE` component, f_{left} as 99th percentile fan-out for the `LEFT_TOKEN` component and n_{total} as total number of nodes in the `COVERAGE` component, the selectivity is calculated by dividing the

Algorithm 5.16 `Inclusion` operator implementation of the `filter(…)` function. It needs the corresponding `LEFT_TOKEN` (g_L), the `RIGHT_TOKEN` (g_R) and the `ORDERING` (g_O) components as arguments. Also, the LHS and RHS node candidates (n_{lhs}, n_{rhs}) are given.

1: **function** FILTER($n_{lhs}, n_{rhs}, g_L, g_R, g_O$)
2: $start_{lhs} \leftarrow$ CONNECTEDTOKEN(n_{lhs}, g_R)
3: $end_{lhs} \leftarrow$ CONNECTEDTOKEN(n_{lhs}, g_L)
4: $start_{rhs} \leftarrow$ CONNECTEDTOKEN(n_{lrs}, g_R)
5: $end_{rhs} \leftarrow$ CONNECTEDTOKEN(n_{rhs}, g_L)
6: $l \leftarrow$ DISTANCE($start_{lhs}, end_{lhs}, g_O$) ▷ span length of LHS
7: $cond_{left} \leftarrow$ ISCONNECTED($start_{lhs}, start_{rhs}, 0, l, g_O$) ▷ path between left-most tokens exists in `ORDERING` component and has maximum length l
8: $cond_{right} \leftarrow$ ISCONNECTED($end_{rhs}, end_{lhs}, 0, l, g_O$) ▷ path between right-most tokens exists in `ORDERING` component and has maximum length l
9: **return** $cond_{left} \wedge cond_{right}$

Algorithm 5.17 `Inclusion` operator implementation of the `retrieveMatches(…)` function. It needs the corresponding`LEFT_TOKEN` (g_L), the `RIGHT_TOKEN` (g_R) and the `ORDERING` (g_O) component as arguments. Also, the LHS source node candidate n_{lhs} is given.

1: **procedure** RETRIEVEMATCHES(n_{lhs}, g_L, g_R, g_O)
2: $start_{lhs} \leftarrow$ CONNECTEDTOKEN(n_{lhs}, g_L, g_O) ▷ get left-most covered token for source
3: $end_{lhs} \leftarrow$ CONNECTEDTOKEN(n_{lhs}, g_R) ▷ get right-most covered token for source
4: $l \leftarrow$ DISTANCE($start_{lhs}, end_{lhs}, g_O$) ▷ span length of source
5: **for** $t \in$ FINDCONNECTED($start_{lhs}, 0, l, g_O$) **do** ▷ find each token which is between the left and right border
6: **emit** t ▷ token itself is included
7: **for** $n \in$ GETOUTGOINGEDGES(t, g_L) **do** ▷ get all left-aligned nodes
8: $end_n \leftarrow$ GETOUTGOINGEDGES(n, g_R)[0] ▷ right-aligned token of candidate
9: **if** ISCONNECTED($end_n, end_{lhs}, 0, l, g_O$) **then** ▷ path between right-most tokens exists in `ORDERING` component and has maximum length l
10: **emit** n

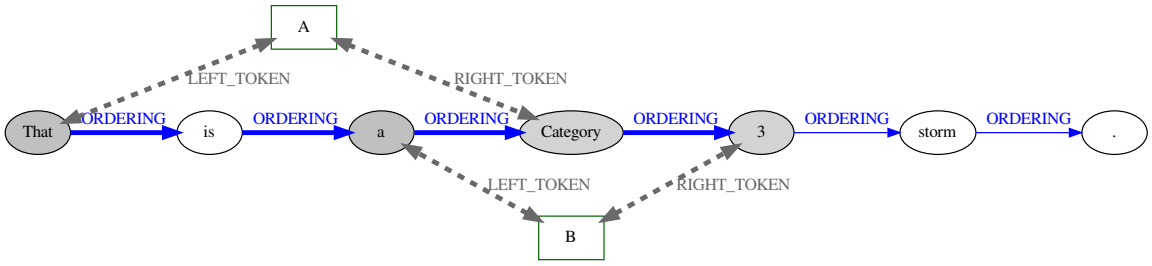


Figure 5.15.: Overlap operator example. The span node A overlaps B because there is a path in the `ORDERING` component from the left-most token of A ("This") to the right-most token of B ("3") and one from the left-most token of B ("a") to the right-most token of A ("Category").

expected number of included nodes per node with the total number:

$$selectivity_{Inclusion}(f_{cov}, f_{left}, n_{total}) = \frac{f_{cov} + (f_{cov} \cdot f_{left})}{n_{total}} \tag{5.9}$$

The fan-out for the `COVERAGE` component f_{cov} is used as an estimate for the covered tokens per node and it is added to the estimation of left-aligned nodes ($f_{cov} \cdot f_{left}$). This estimation does not take the span lengths into account and assumes every left-aligned node is an included node, which is a pessimistic approximation.

5.6.5. Overlap operator

Two nodes overlap, if the intersection of their sets of covered tokens is not empty. For instance in Figure 5.15 both nodes A and B cover the "a" and the "Category" tokens. For just filtering pairs of nodes in the `Overlap` operator (`_o_`) implementation, calculating the set of all covered tokens would be too expensive. Instead, the `filter(…)` function checks if there is a path in the `ORDERING` component from the left-most covered token from node x to the right-most token of node y and another one from the left-most covered token of y to the right-most covered token of y. See Algorithm 5.18 for pseudo-code. The implementation of the `retrieveMatches(…)` function needs to find all overlapped nodes, which includes all covered tokens, and thus uses a different approach which is described in Algorithm 5.19. First, all covered tokens for the start node are retrieved. These can be found by getting all outgoing edges of the `COVERAGE` component (line 6). For each covered token, the outgoing edges in the `INVERSE_COVERAGE` component are used to find all non-token nodes that are covering this token (line 8). Since these nodes cover the same tokens, they fulfill the overlap criterion. This approach can yield duplicate results and a set is used to filter out these duplicates (line 11).

Calculating the selectivity for the `Overlap` operator is similar to the `Inclusion` operator. It takes the 99th percentile value for the fan-out of both the `COVERAGE` component (f_{cov}) and the `INVERSE_COVERAGE` component (f_{icov}) as argument in addition

Algorithm 5.18 `Overlap` operator implementation of the `filter(…)` function. It needs the corresponding `LEFT_TOKEN` (g_L), the `RIGHT_TOKEN` (g_R) and the `ORDERING` (g_O) components as arguments. Also, the LHS and RHS node candidates (n_{lhs}, n_{rhs}) are given.

1: **function** FILTER($n_{lhs}, n_{rhs}, g_L, g_R, g_O$)
2: $start_{lhs} \leftarrow$ CONNECTEDTOKEN(n_{lhs}, g_R)
3: $end_{lhs} \leftarrow$ CONNECTEDTOKEN(n_{lhs}, g_L)
4: $start_{rhs} \leftarrow$ CONNECTEDTOKEN(n_{lrs}, g_R)
5: $end_{rhs} \leftarrow$ CONNECTEDTOKEN(n_{rhs}, g_L)

6: $cond_{lhs} \leftarrow$ DISTANCE($start_{lhs}, end_{rhs}, g_O$) $>= 0$ ▷ path between LHS left-most token and RHS right-most token exists in `ORDERING` component
7: $cond_{rhs} \leftarrow$ DISTANCE($start_{rhs}, end_{lhs}, g_O$) $>= 0$ ▷ path between RHS left-most token and LHS right-most token exists in `ORDERING` component
8: **return** $cond_{lhs} \wedge cond_{rhs}$

Algorithm 5.19 `Overlap` operator implementation of the `retrieveMatches(…)` function. It needs the corresponding `COVERAGE` (g_C) and the `INVERSE_COVERAGE` (g_{IC}) component as arguments. Also, the LHS source node candidate n_{lhs} is given.

1: **procedure** RETRIEVEMATCHES(n_{lhs}, g_C, g_{IC})
2: $R \leftarrow \emptyset$ ▷ use set to filter out duplicates
3: **if** ISTOKEN(n_{lhs}) **then**
4: $C \leftarrow \{n_{lhs}\}$
5: **else**
6: $C \leftarrow$ FINDCONNECTED(n_{lhs}, g_C)
7: **for** $t \in C$ **do** ▷ all covered token
8: **for** $n \in$ GETOUTGOINGEDGES(t, g_{IC}) **do**
9: $R \leftarrow R \cup \{n\}$ ▷ all nodes covering this token
10: $R \leftarrow R \cup \{t\}$ ▷ also token itself
11: **for** $n \in R$ **do**
12: **emit** n ▷ output the unique results

to the total number of nodes n_{total}:

$$selectivity_{Overlap}(f_{cov}, f_{icov}, n_{total}) = \frac{f_{cov} + (f_{cov} \cdot f_{icov})}{n_{total}} \tag{5.10}$$

As for the `Inclusion` operator, the fan-out of the `COVERAGE` component f_{cov} is used as an estimate for the covered tokens per node. Instead of estimating the left-aligned nodes, the product of $(f_{cov} \cdot f_{icov})$ is used as an estimation for the number of non-token nodes covering the tokens. The sum of both is then used as estimation for the number of overlapped nodes per source node.

5.6.6. Metadata search

Searching for metadata is implemented by introducing a new operator `PartOfSubCorpus`. AQL allows filtering results of a query with the special `meta::` term (see Section 3.2.4), but does not expose a binary operator that connects annotation nodes with the ones from the corpus graph (see Section 4.2.1 for a description and example). Like the `Dominance` and `Pointing` operators, `PartOfSubCorpus` inherits its implementation for the `filter(…)` and `retrieveMatches(…)` functions from `AbstractEdgeOperator`, which is described in Section 5.6.1. The operator allows finding nodes that are connected in the `PART_OF_SUBCORPUS` component (which is also described in Section 4.2.1 in more detail) by paths of arbitrary length.

Selectivity estimation has been adjusted for the special layout of the corpus graphs and typical queries. The `PART_OF_SUBCORPUS` component is hierarchical with the following properties:

- the roots of the graph are always annotation nodes,
- the second "layer" of the graph are documents, and
- other nodes are corpora or sub-corpora.

Queries usually start by finding matching documents for an annotation node. This means, that the path length is not really important because the documents are always connected to the annotation nodes by paths with length 1. Also, the average fan-out is misleading, because the number of output edges for non-annotation nodes is not really relevant. Thus, the number or reachable nodes for a single source node is estimated with the maximum fan-out f_{max} of the component:

$$selectivity_{PartOfSubCorpus}(f_{max}, n_{total}) = \frac{f_{max}}{n_{total}} \tag{5.11}$$

5.7. Joins and filters

GraphANNIS implements two different kinds of joins, which are used depending on the structure of the query plan. The general `NestedLoopJoin` join implements a nested loop join that can take any other physical execution node as both LHS and RHS. See Algorithm 5.20 for the implementation. The nested loop implementation takes the

Algorithm 5.20 `NestedLoopJoin` implementation. It takes the operator *op* as predicate and two iterators (it_{outer}, it_{inner}) as argument. The iterators correspond to the LHS and RHS of the join, but they can be switched if necessary. This nested loop implementation materializes the results of the inner iterator after the first run. Thus, if the inner iterator contains complex annotation searches or joins, these are not executed twice.

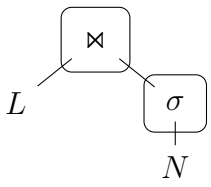
```
1:  procedure NESTEDLOOPJOIN(op, it_outer, it_inner)
2:      materalized ← false
3:      m ← emptyList                    ▷ init materialized inner tuples as empty list
4:      for i ∈ it_outer do
5:          if materialized then
6:              for j ∈ m do
7:                  if FILTER(op, i, j) then              ▷ apply filtering by operator
8:                      emit (i, j)
9:          else
10:             for j ∈ it_inner do
11:                 INSERT(m, j)                  ▷ insert inner to materialized list
12:                 if FILTER(op, i, j) then          ▷ apply filtering by operator
13:                     emit (i, j)
14:         materialized ← true      ▷ switch to materialized list after first iteration
```

Algorithm 5.21 `IndexJoin` implementation. It takes the operator *op*, the LHS iterator it_{lhs} and the annotation selection filter function f_{rhs} for the RHS as argument.

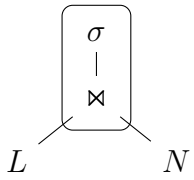
```
1:  procedure INDEXJOIN(op, it_lhs, f_rhs)
2:      for i ∈ it_lhs do
3:          for j ∈ RETRIEVEMATCHES(op, i) do      ▷ find all RHS candidates that
    fulfill the operator condition
4:              if f_rhs(j) = true then      ▷ filter candidates by annotation condition
5:                  emit (i, j)
```

outer and inner iterator as arguments corresponding to the LHS and RHS iterators. They can be switched if necessary. The nested loop join will materialize the result of the inner iterator, to avoid the re-execution of complex annotation searches or joins by the inner side iterator.

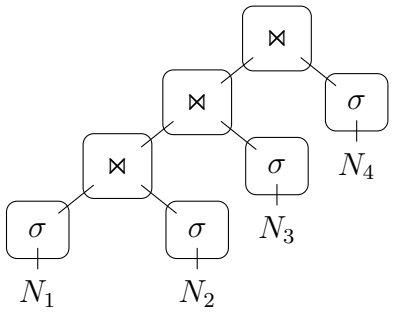
In case that the RHS is an annotation search like



the alternative `IndexJoin` implementation can be used, because the plan can be rewritten to apply the join first on the node relation N and then apply the annotation selection operation as part of the same execution node:

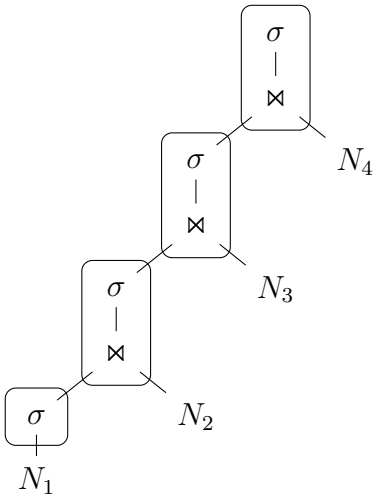


The pseudo-code implementation of the `IndexJoin` is given in Algorithm 5.21. Instead of applying the `filter(…)` function like the nested loop join, the `IndexJoin` retrieves all possible RHS candidates for a specific LHS item by calling the `retrieveMatches(…)` function of the given operator on the LHS item. This output is then filtered with the node annotation search condition.[21] Since the `retrieveMatches(…)` implementation is typically supported by indexes on the graph, this index-lookup join can be more efficient than the `NestedLoopJoin`, depending on the number of the matches retrieved by the annotation search versus the ones that match the AQL operator definition. The query planner will locally re-write a `NestedLoopJoin` to an `IndexJoin` if possible. Since the RHS of an `IndexJoin` never contains another join, this strategy will produce left-deep join trees (Garcia-Molina et al. 2000, pp. 395 ff.). For instance, in the following execution plan



all joins can implemented as `IndexJoin` because the RHS is always directly connected to a selection on a node relation and the query plan can be rewritten into:

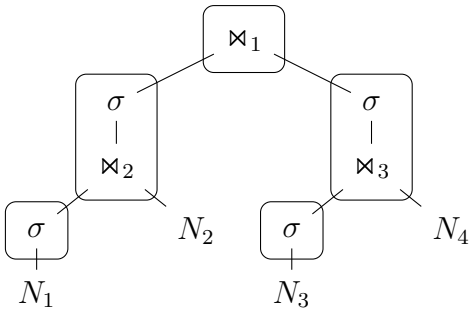
[21] It could be argued that it is much more flexible to add a separate execution node, that filters results by the node annotation search condition. In AQL queries, the operands of an operator always reference the node annotation relation directly. If the AQL operator condition is used as the predicate of the join, it is a general pattern that the selection for the node annotation search should be applied directly thereafter. Since this pattern is so common, it was decided to include the filter into the execution node implementation of the `IndexJoin`.



Preferring left-deep join trees reduces the number of possible plans that have to be evaluated when comparing their cost (see Section 6.2.1 for a description of the cost-based planning). Creating a left-deep join tree is not always possible when multiple AQL operators refer to the same nodes as their LHS and RHS. For example, in the query

```
node & node & node & node & #1 . #2 & #3 . #4 & #1 . #4
```

the last operator `#1 . #4` references an LHS and RHS that have already been used in other parts of the query. This query would result in a query plan that is not a left-deep join tree:



Only the joins $\bowtie_2$ and $\bowtie_3$ can be implemented as `IndexJoin` whereas join $\bowtie_1$ must be implemented as `NestedLoopJoin`. Parallelized implementations exist for both join types and are described in more detail in Section 6.3.

In other cases, where AQL queries have more than one operator defined between the exact same two annotation searches, applying a join does not make sense because no new combinations of tuples will be generated. In this case, the output of the first join is filtered with help of the `BinaryFilter` class. It takes an iterator and an operator as argument and applies the `filter(…)` function to each item of the iterator.

5.8. Database management and serialization

The central class that contains the string storage, the annotation storage and the different graph storages is the `DB` class. See Figure 5.16 for a class diagram of `DB`

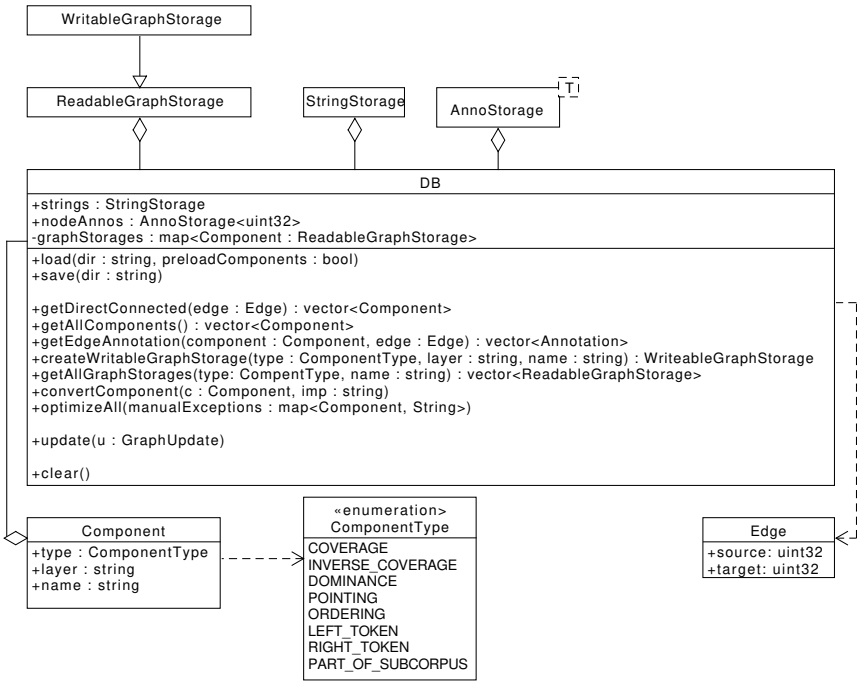


Figure 5.16.: Class diagram of the `DB` and related classes.

and its related classes. Every `DB` instance has a string storage and an annotation storage. Different graph storages are part of the `DB`, depending on the annotations that are present in the specific corpus (see Section 4.2 for a detailed description how different linguistic annotations are represented using graph storages). It is possible to change the implementation of a graph storage component by asking the `DB` instance to make it either a `WriteableGraphStorage` or by converting a component to its optimized implementation. An important function of the `DB` is to store and load the corpora. This allows persisting optimized binary representations on disk and load them when necessary. Serialization is done with the help of the Cereal library and each graph storage implementation is serialized separately. Instead of using a general graph-based exchange format, the internal data structures of the specific graph storage implementation are persisted onto the disk, which allows faster loading. Also, it is possible to only load the string storage and annotation storage into memory. In this case, graph storages are loaded from disk whenever needed by a query. This enables memory saving in a server scenario where multiple corpora could be loaded, but not all graph storages are needed all the time.

Another important management class is the `CorpusStorageManager`, whose class diagram is given in Figure 5.17. The `CorpusStorageManager` is responsible for organizing a collection of `DB` instances (where each `DB` instance contains one corpus) and to provide an external API for management and querying of these corpora. Querying functions are

- counting the number of results for a query (which is given as JSON),
- finding matches and returning a list of the IDs of the matched annotations,

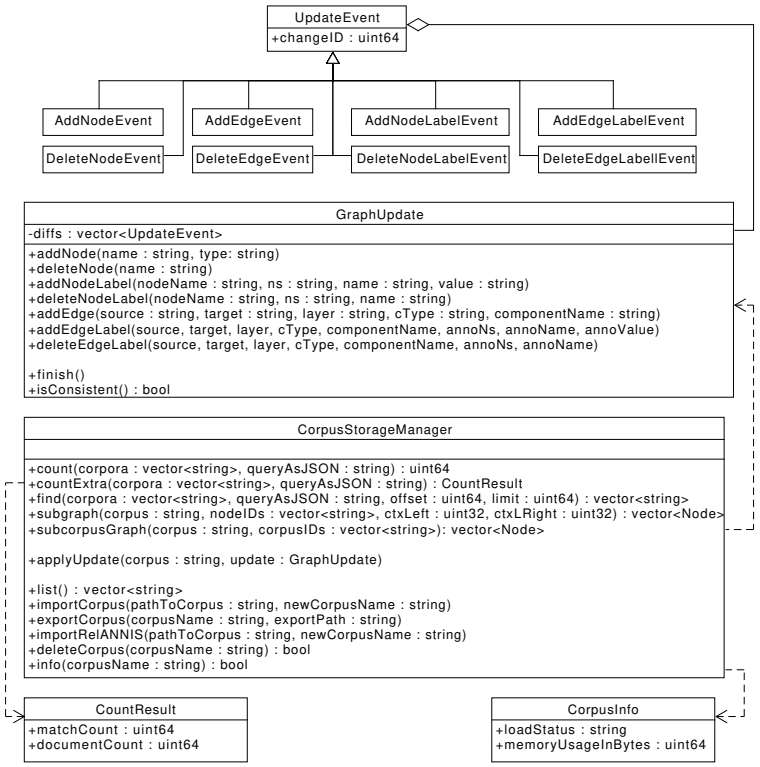


Figure 5.17.: Class diagram of the `CorpusStorageManager` and related classes.

- obtaining the subgraph for a match within its context, and
- obtaining the complete subgraph for a document.

A `CorpusStorageManager` instance is configured to use a specific disk folder to persist the corpora it manages and it loads and unloads corpora automatically as needed. It can be configured to use a maximum amount of main memory and will unload corpora when the limit is reached. Corpora can also be imported from or exported to other locations on the file system. In addition, it is possible to import corpora that are encoded in the legacy relANNIS format. Updates to existing corpora can be performed with the `applyUpdates(…)` function which takes a `GraphUpdate` instance as argument. A `GraphUpdate` is a list of update events (implemented by the different sub-classes of the `UpdateEvent` class) which will be applied to the corpus atomically and durably. This is ensured by using a write-ahead log which contains these updates and is used to restore a transaction when the program is exited before the regular `DB` serialization is completed. All functionality of the `CorpusStorageManager` is also wrapped in a Java API which allows integrating it into the existing Java-based web-service.

6. Optimization and parallelization

In the previous chapter, the main components of the graphANNIS query execution engine have been described. The present chapter will go into more detail about the different optimization techniques that are used in graphANNIS to make the query system faster. Some of these optimizations are typical for relational database systems while others are more specific to the graph-based model of graphANNIS. In addition to these optimization techniques, this chapter also describes an approach to parallelize the execution of joins.

6.1. Automatic selection of graph storage implementations

GraphANNIS is intended to cope with different kind of annotations. These annotations share their graph-based representation, but they result in different types of graphs. For each type of graph, a different graph storage implementation is optimal in regard to the execution times of its reachability query functions given any possible argument (see Section 5.5 for a description of these functions). Since a corpus is typically non-volatile, it is possible to perform a detailed analysis and use the statistics as described in Section 5.5 to choose a graph storage implementation for each component. The following rules (ordered in the same order as they are applied) are used to choose the implementation:

1. If the longest path of the component has length ≤ 1, use the `AdjacencyListStorage`.
2. If the component is a tree and the maximum number of outgoing edges per node is ≤ 1, use the `LinearStorage`.
3. If the component is a tree or a DAG and the average number of visits for each node in a DFS traversal is ≤ 1.03, use the `PrePostOrderStorage`.
4. In any other case use the `AdjacencyListStorage`.

These rules are independent of any query workload and only analyze the structure of the component. Rule 1 favors adjacency lists for simple types of graphs, where there are only single edges, but no longer paths between nodes. Since adjacency lists store all outgoing edges for a node in a map, finding reachable nodes can be implemented by just checking the outgoing edges directly. Thus, the adjacency lists provide a fast implementation, while having a low overhead compared to the other implementations. The `LinearStorage` is chosen by Rule 2 because a component

annotation	type	max. path length	is tree	max. out. edges	avg. DFS visits	implementation
token precedence	`ORDERING`	1866	✓	1	1.0	`LinearStorage`
spans	`COVERAGE`	1		1867	8.43	`AdjacencyListStorage`
dependency tree	`POINTING`	16	✓	14	1.0	`PrePostOrderStorage`
co-referents	`POINTING`	94		5	1.45	`AdjacencyListStorage`
constituent tree	`DOMINANCE`	29	✓	15	1.0	`PrePostOrderStorage`
RST	`DOMINANCE`	22		97	1.18	`AdjacencyListStorage`

Table 6.1.: Graph storage implementations as chosen by the heuristic for exemplary annotations of the GUM corpus (Zeldes 2016b). These are the same annotation types that have been used in the example Figure 1.1.

with these properties only consists of linear paths, which is the exact use-case the `LinearStorage` was designed for. Typically, the `ORDERING` component (see Section 4.2.2 for an explanation) is using a `LinearStorage`, and this rule helps to accelerate AQL operators which involve text coverage. For components that fulfill Rule 3, the pre-/post-order encoding is chosen, because it is optimized for computing reachability in trees. Duplication of nodes has been an issue for applying the pre-/post-order encoding to DAGs, but in graphANNIS, there is a statistical value which allows measuring this overhead explicitly. This overhead estimation is used to decide whether the `PrePostOrderStorage` implementation is used for DAGs, with a fixed limit of 3 percent overhead. In the future, this limit should be configurable, so a user can balance the main memory usage and performance on its own. Rule 4 defines the `AdjacencyListStorage` as fall-back implementation for unknown types of graphs. Table 6.1 shows some example annotations and which graph storage implementation is chosen by this heuristic.

6.2. Query optimizer

Several alternative execution plans can be generated for a given AQL query. This section describes how the best execution plan is chosen in graphANNIS. The selection of the best plan is performed by the `SingleAlternativeQuery` class and can be configured to either apply or disregard specific optimizations.

6.2.1. Result size and cost estimation

In order to compare different execution plans, a cost function is executed, and the plan with a smaller cost is treated as the better one. GraphANNIS uses a very simple cost-model, using the number of overall processed tuples in an execution plan as cost. This model only needs estimations of result sizes and intermediate result sizes and omits more complex estimations like costs for CPU-instructions or main memory access costs.

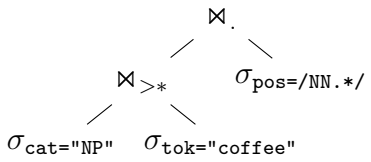
Consider the exemplary query, which finds instances of the word “coffee” directly followed by another noun which is part of the same phrase:

Example 6.1

```
cat="NP" & tok="coffee" & pos=/NN.*/
        & #1 >* #2 & #2 . #3
```

The following execution plan could be generated (with simplified notation for the selection operation):

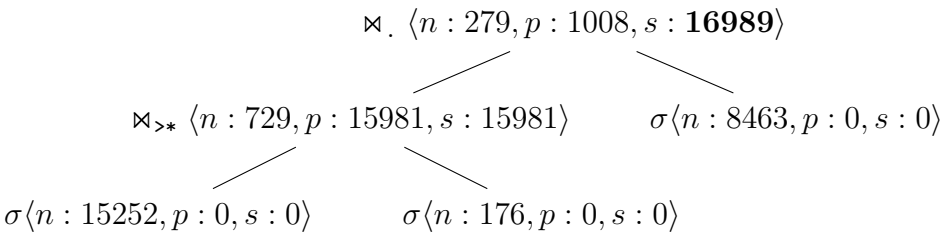
Example 6.2



To each node of this execution plan, three estimations are assigned:

- the estimated number of output tuples n,
- the estimated number of tuples processes inside the execution step itself p, and
- the estimated sum of processed tuples for the subtree s.

For the previous example, an example cost estimation could be the following:



The first join $\bowtie_{>*}$ combines two selections which have an estimated output size of 15252 and 176 respectively These two selection node iterators are not processing tuples by themselves, according to the cost estimation, but only provide an estimation for the number of output tuples which is calculated using the equi-depth histogram of the annotation storage (see Section 5.4). Based on the selectivity function $selectivity_X(\dots)$ of the specific AQL operator and the cross product of the output sizes of LHS and RHS, the output size of this join is estimated to be

$$selectivity_X(\dots) \cdot (n_{lhs} \cdot n_{rhs}) \tag{6.3}$$

In this example, this results in an output size of the join of 729. The join needs to process $p = 15981$ tuples and is the LHS input for the parent join $\bowtie_{.}$. For this parent join, the selectivity is used to estimate the number of output tuples and the number of processed tuples. This time the sum of processed tuples for the whole subtree ($s = 16986$) is larger than the sum of tuples processed inside the join itself because the processed tuples of the join $\bowtie_{>*}$ are added. Since $\bowtie_{.}$ is the root of the execution plan, its sum of processed tuples ($s = 16986$) is used as the cost of the whole plan.

Estimating the number of processed tuples is performed differently for the two types of joins. For the `NestedLoopJoin` the text-book function

$$n_{lhs} + (n_{lhs} \cdot n_{rhs}) \tag{6.4}$$

is used (Garcia-Molina et al. 2000, p. 278). The `IndexJoin` will also process each item of the LHS once, but for each LHS tuple it will only process the number of reachable nodes as defined by the AQL operator. The average number of reachable nodes can be estimated by using the selectivity function of the operator multiplied by an estimation for the average number of possible output nodes in total, leading to a more simplified estimation formula:

$$n_{lhs} + (n_{lhs} \cdot averageReachable) \tag{6.5}$$

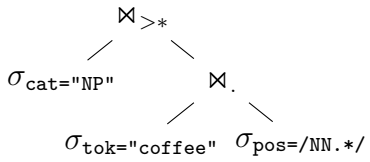
$$= n_{lhs} + \left(n_{lhs} \cdot \frac{selectivity_X(\dots) \cdot (n_{lhs} \cdot n_{rhs})}{n_{lhs}} \right) \tag{6.6}$$

$$= n_{lhs} + (selectivity_X(\dots) \cdot (n_{lhs} \cdot n_{rhs})) \tag{6.7}$$

In this cost model, the `IndexJoin` will always be preferred to a `NestedLoopJoin` if the structure of the execution plan allows its usage because the selectivity for an operator can not be larger than 1. The worst-case cost of an `IndexJoin` is the same as the cost for a `NestedLoopJoin`, but as soon as the operator selectivity is less than 1 (which is typically true), the `IndexJoin` is assumed to be more efficient.

6.2.2. Optimizing join order

Execution plans, as they have been described in Section 5.2, are created by adding each binary AQL operator and creating a join between either the referenced annotations searches or the output of a previous join that includes the referenced annotation search.[1] For instance, the AQL query in Example 6.1 has two binary operators. By first applying the `#1 >* #2` and then the `#2 . #3` operator, the original plan from Example 6.2 would be generated. If the operators are applied in reverse order, another execution plan is generated:



In the current implementation, finding the optimal join order is achieved by finding the optimal order for adding the AQL operators to the plan. If the number of AQL operators in a query is at most 6, all permutations of possible orders are calculated, the cost is calculated for each permutation and the plan with the smallest cost is selected. Since the number of plans gets too large and calculating the cost for all plans is too expensive, for queries with more than 6 operators a simple heuristic optimization algorithm has been developed. See Algorithm 6.1 for its pseudo-code.

[1] If both annotation searches are already included in other joins, a filter is applied instead of a join.

Algorithm 6.1 Pseudo-code for the heuristic operator order optimizer.

```
1:  procedure OperatorOrder(ops_initial)
2:      unsuccessful ← 0
3:      ops_best ← ops_initial
4:      cost_best ← cost(order_best)
5:      while unsuccessful < 5 · |operators_initial| do      ▷ limit maximum attempts
6:          family ← emptyList      ▷ avoid local minima with multiple generations
7:          add(family, ops_best)                          ▷ best order is used as ancestor
8:          for i ← 1, 4 do
9:              tmp ← family[i − 1]                        ▷ use previous generation as basis
10:             (a, b) = randomPair(tmp)
11:             tmp = swap(tmp, a, b)                      ▷ randomly swap two operators
12:             add(family, tmp)                                          ▷ add to family
13:         unsuccessful ← unsuccessful + 1
14:         for i ← 1, 4 do
15:             if cost(family[i]) < cost_best then        ▷ found better operand order
16:                 unsuccessful ← 0
17:                 ops_best = family[i]
18:                 cost_best ← cost(family[i])
```

The general idea of the heuristic optimizer is to switch pairs of operators randomly. This switch is performed several times until no improved plan could be found. The number of maximum tries depends on the number of operators in the AQL query. In order to avoid local minima, multiple mutated generations are created from the same original ancestor.

6.2.3. Query rewriting rules

Like other search systems, graphANNIS rewrites the queries provided by the user in order to allow faster execution without changing the semantics of the query. These rules are applied before the join order optimization is executed.

Switching operands for commutative operators

For operators which are commutative (see 5.11 for the logical properties of each operator), the LHS and RHS can be changed without changing the semantics of the query. This is useful for queries where the estimated number of output nodes for one size is considerably smaller than for the other side. In order to minimize the number of processed tuples, the annotation search with the smaller output size is always used as LHS for commutative operators and the query is rewritten with switched operands if needed.

Improper used regular expressions

Another frequent problem in user-provided queries is the unnecessary use of the regular expression search. Users indicate that they want to apply a regular expression by writing the annotation search in the form `anno=/value/`. However, we often encountered queries where such expressions do not contain any place-holder or other characters with a special meaning for regular expressions. In the case that no regular expression meta-characters exist, the query parser will replace the regular expression with a constant annotation value search. Searches that match every possible string (for example, the regular expression `.*`) are less frequently observed, but still a relevant problem. This will lead to a processing overhead in the annotation search because the evaluation of the regular expression for each match will still take a considerable amount of time. Thus, when creating the execution plan it is checked if a regular expression is unbound (which means it matches every string) by using the appropriate function of the external regular expression library RE2. If this is the case, it will be replaced with a search for the annotation key only (which is implemented by the `ExactAnnoKeySearch` class described in Section 5.4).

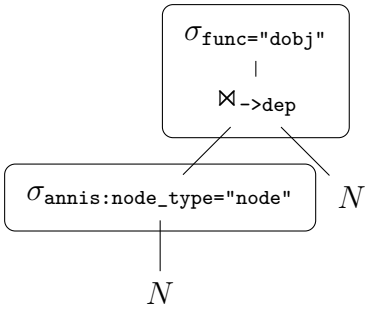
Replace low selectivity annotation searches with edge annotation search

One of the three node search iterators `ExactAnnoKeySearch`, `ExactAnnoValueSearch`, and `RegexAnnoSearch` is normally used to implement the leaf nodes of an execution plan. In some cases, especially when a node annotation search is only limited on the annotation key, the selectivity of these node searches is very low. If the annotation search is referenced by an operator which inherits the `AbstractEdgeOperator` class (for example, the pointing and dominance relation operators), it is possible to exchange the node annotation search with an instance of the `NodeByEdgeAnnoSearch` iterator.

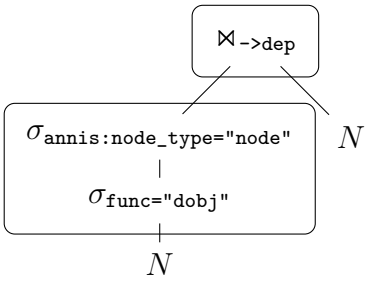
For example, the query

```
node ->dep[func="dobj"] node
```

will result in the following initial plan with two execution nodes:



One execution node selects all annotation nodes of the graph by executing an `ExactAnnoValueSearch` on the `node_type` label, which is present for all nodes. The other execution node finds all nodes that match the operator definition and performs an additional selection on the edge annotation condition. In contrast to the selection of all nodes, the edge annotation condition is very selective. In such a case, the `NodeByEdgeAnnoSearch` iterator is used instead of the original node annotation search:



It returns all nodes that have an outgoing edge in a specific component and optionally a specific edge annotation. An additional selection for the node annotation is applied as part of the `NodeByEdgeAnnoSearch` iterator to select the nodes matching the original annotation search definition. Instances of the `AbstractEdgeOperator` have a special `guessMaxCountEdgeAnnos(…)` function which allows estimating the number of nodes returned by this iterator. When an execution plan is created, this value is compared to the estimated number of results of the node annotation search, and if it is lower, the plan is rewritten to use `NodeByEdgeAnnoSearch`.

6.3. Parallelization of joins

Modern CPUs provide features to parallelize program execution. In certain situations, graphANNIS will make use of this possibility to accelerate the response time. Joins are an attractive optimization target for parallelization because they process many tuples. Both thread-based and "Single instruction, multiple data" (SIMD) CPU instructions have been used in graphANNIS to optimize joins. These parallelization strategies are limited to a single computer: graphANNIS does not currently support distributed execution on different host machines that are connected, for example, over a network.

6.3.1. Thread-based implementation of joins

GraphANNIS has thread-based implementations of both `NestedLoopJoin` and `Index Join`. They use the same implementation idea as the non-thread versions (see Algorithm 5.20 and 5.21), but add a parameter p to allow parallel execution in p threads. When using threads and shared memory for parallelization, the gain of parallel computation must be balanced against the overhead of the thread management and synchronization (Cormen et al. 2009, p. 773). Thread management encompasses both requesting a new thread from the operating system and the thread scheduling that the operating system needs to perform. These overhead costs become larger the more threads are created and run in parallel. To keep management cost low, the threaded versions of the joins in graphANNIS use a shared thread-pool, which avoids re-creation of threads, and limits the number of parallel threads to the number of CPUs of the system. Extending the existing joins to run in parallel via threads can be achieved without adapting the underlying `filter(…)` and `retrieveMatches(…)` functions. Also, it is important to ensure that the tasks that are run in parallel are not too short, so that the overhead introduced by the synchronization is justified.

Both parallel joins need to synchronize the access to their output values, which is ensured with a simple shared queue implementation.

Algorithm 6.2 shows the pseudo-code for `ThreadNestedLoopJoin`. It spawns p threads at the beginning (line 5). Each thread consists of a loop over each possible pair of LHS and RHS iterator results (line 7), which are filtered with the operator (line 8). This means that each call to the `filter(…)` function of the AQL operator implementation is executed in parallel. Access to the iterators requires being synchronized, which is secured by a lock in the function which generates the next pair.

The `ThreadIndexJoin` (Algorithm 6.3) also initializes p threads at the beginning and starts a loop function for each thread. This loop retrieves the next LHS item from the iterator (line 5), and only this access is synchronized over all threads. As a result, calling the `retrieveMatches(…)` function of the AQL operator implementation (line 6) and checking the RHS annotation (line 7) is executed in parallel for each LHS item, which increases the amount of work that is actually parallelized in comparison to the parallel nested loop join.

It must be decided how many threads a particular join will use in case of a query having multiple joins. This is done by using the best non-parallelized plan as the basis and by assigning threads to joins statically at planning time. The execution plan contains the number of processed tuples for every single join, and this number is used to identify the joins with the highest potential for parallel execution. Two threads are assigned to the join with the highest number of processed steps, this number of processed steps is divided by two, the plan is updated with this new number and these steps are repeated until all threads have been assigned. Assigning threads pair-wise guarantees that there are always at least two threads for a parallelized join and no join has both the synchronization overhead and only one thread assigned.

6.3.2. SIMD-based implementation of joins

While recent compilers are able to generate vectorized instructions automatically, auto-vectorization can be limited and may need adjustments to the structure of the code (Kretz 2015, pp. 19 ff.). Manual usage of SIMD features of CPUs gives more control over the parallelization process. Joins process many tuples and applying SIMD to joins in graphANNIS could help improving execution speed. A part of the join implementation that can be represented as vector operation is needed in order to use SIMD.

Such a vector operation is the loop in line 3 of the `IndexJoin` (Algorithm 5.21) which iterates over all candidate nodes returned by the `retrieveMatches(…)` function and filters them according to the annotation condition. If the annotation condition is a static annotation value (which is expressed as the numeric ID of a string) and not a regular expression, this loop can be replaced by a vector instruction that checks several IDs in parallel and returns which of the IDs match a given template. This optimized join was implemented as a separate `SIMDIndexJoin` class. The vector instructions are realized with the help of the Vc library (Kretz 2015), which provides abstractions for different kind of SIMD instruction sets of different types of CPUs and automatically chooses the best available instruction set at runtime. Candidate nodes and their annotation values are collected in vectors with the appropriate length for the SIMD

Algorithm 6.2 `ThreadNestedLoop` implementation. It takes the operator *op* as predicate, two iterators (it_{outer}, it_{inner}) and the number of parallel threads p as argument. The iterators correspond to the LHS and RHS of the join, but they can be switched if necessary. This nested loop implementation materializes the results of the inner iterator after the first run. Thus, if the inner iterator contains complex annotation searches or joins, these are not executed twice. Parallelization is achieved by executing a loop in each thread, which first fetches the next pair of match candidates and then applies `filter(…)` on each of these pairs. Only one thread can access the iterators and the materialized array at the same time.

1: **procedure** THREADNESTEDLOOPJOIN($op, it_{outer}, it_{inner}, p$)
2: $m \leftarrow emptyArray$ ▷ init materialized inner tuples as empty array
3: $m_{idx} \leftarrow \emptyset$ ▷ empty value for current index of materialized array
4: **for all** $t \in [1, p]$ **do**
5: **spawn** LOOP($op, it_{outer}, it_{inner}, m, m_{idx}$) ▷ run function in p threads

6: **procedure** LOOP($op, it_{outer}, it_{inner}, m, m_{idx}$)
7: **while** $(i, j) \leftarrow$ NEXTPAIRSYNCHRONIZED($it_{outer}, it_{inner}, m, m_{idx}$) **do**
8: **if** FILTER(op, i, j) **then** ▷ apply filtering by operator
9: **emit** (i, j)

10: **function** NEXTPAIRSYNCHRONIZED($it_{outer}, it_{inner}, m, m_{idx}$)
11: **begin lock** ▷ it_{outer}, it_{inner}, m and m_{idx} can be only accessed by one thread at the same time, lock is released when function returns
12: **while** PEEK(it_{outer}) $\neq \emptyset$ **do** ▷ get next element without consuming it
13: $i \leftarrow$ PEEK(it_{outer})
14: **if** $m_{idx} \neq \emptyset$ **then** ▷ use materialized inner
15: **if** $m_{idx} < |m|$ **then**
16: $j \leftarrow m[m_{idx}]$ ▷ get next inner from materialized array
17: $m_{idx} \leftarrow m_{idx} + 1$
18: **return** (i, j)
19: **else**
20: $j \leftarrow$ NEXT(it_{inner}) ▷ get next inner from iterator
21: **if** $i \neq \emptyset$ **then**
22: INSERT(m, j) ▷ append inner to materialized list
23: **return** (i, j)
24: $m_{idx} \leftarrow 0$ ▷ inner was completed, use materialized array from now on
25: NEXT(it_{outer}) ▷ consume next element of outer iterator
26: **return** $\emptyset$ ▷ both iterators are exhausted

Algorithm 6.3 `ThreadIndexJoin` implementation. It takes the operator *op*, the LHS iterator it_{lhs}, the annotation selection filter function f_{rhs} for the RHS and the number of parallel threads p as argument.

```
1: procedure threadIndexJoin(op, it_lhs, f_rhs, p)
2:     for all t ∈ [1, p] do
3:         spawn loop(op, it_lhs, f_rhs)                    ▷ run function in p threads

4: procedure loop(op, it_lhs, f_rhs, p)
5:     while i ← nextSynchronized(it_lhs) do                ▷ get next LHS
6:         for j ∈ retrieveMatches(i) do    ▷ find RHS candidates by operator cond.
7:             if f_rhs(j) = true then      ▷ filter candidates by annotation condition
8:                 emit (i, j)

9: function nextSynchronized(it_lhs)
10:    begin lock ▷ it_lhs can be only accessed by one thread at the same time, lock is released when function returns
11:    return next(it_lhs)
```

instruction set of the host system. These vectors are then compared to a template vector, and the resulting mask vector is used to decide which nodes are included in the result.

This approach only parallelizes a small amount of the work, and there is overhead by copying the candidate annotation values into the vector and by collecting the results. However, it could be implemented in addition to the thread-based joins since it does not need any more CPU cores. Currently, there is no combined implementation of a join that is both thread-based and uses the explicit SIMD instructions. However, in execution plans that have multiple joins, the ones that have no threads assigned are replaced with a SIMD-based one if parallel query execution is enabled and SIMD is available on the host system.

7. Evaluation

In the previous chapters, the design, implementation and various optimization strategies of graphANNIS have been described. These optimizations have been motivated by inspecting the shortcomings of the legacy application, which is based on a relational database, and by applying new ideas for storing the graph-based linguistic annotations and by efficiently retrieving reachable annotation nodes inside their partitioned components. In this chapter, graphANNIS is evaluated against a realistic workload. First, the execution times for this workload are compared with the legacy relational database implementation. Second, a more detailed analysis of the different optimization strategies and their impact on the overall performance of the system is performed. Also, requirements regarding main memory usage are examined.

7.1. Comparing the relational database implementation with graphANNIS

One of the main goals of this work is to design a linguistic query system that is faster than existing solutions. Different query systems often use their own query language, which makes it difficult to perform a large-scale comparison of non-synthetic queries. GraphANNIS is an implementation of the ANNIS Query Language (AQL). Thus only query systems that support AQL can be compared without translating the queries of the workload. Since there already exists an implementation of AQL based on a relational database, it is obvious to use this system as the baseline for graphANNIS. RelANNIS does not only implement AQL, but it is also used as a production system. Several public server installations exist[1] and are used for actual linguistic studies. Thus, it is possible to collect queries on a set of corpora from such a server and use these queries and corpora as a realistic representation of a typical workload.

7.1.1. Workload and the experimental setup

For this evaluation, queries were collected from two publicly accessible servers, one located at the Humboldt-Universität zu Berlin[2] and one at the Georgetown University[3]. The former one collected queries over five periods of time between 2015 and 2017 with a total coverage of about 100 days. The latter one collected queries from November 2014 to August 2016. Queries and corpora have been included in the workload by to the following criteria:

[1] See for example Section A.2 in the appendix for a list of known ANNIS server installations.
[2] `https://korpling.german.hu-berlin.de/annis3/` (last accessed 2017-12-07)
[3] `https://corpling.uis.georgetown.edu/annis/` (last accessed 2017-12-07)

Availability Only corpora which are available under an Open Access license or which at least can be used in an academic context for free have been included in the workload. This ensures that the benchmark can be re-used by others, either to reproduce the results of this work or for comparing other implementations of AQL. Hopefully, a benchmark workload that is publicly available fosters the development of new linguistic query systems and the evaluation of different approaches in designing such systems. A famous example of such an influence of benchmarks is the Transaction Processing Performance Council (TPC), which "[…] had a significant impact on the industry and expectations around benchmarks" (Nambiar et al. 2009, p. 2).

Representability Since AQL is targeted on multi-layer corpora, it must be assured that different kind of annotations and corpora are included in the workload. This selection has been made manually, based on the experience with the relANNIS system. Since new types of annotations and corpora are continuously developed, it will be necessary to extend the selection of corpora in the future.

Query language features supported by graphANNIS Only queries that exclusively use features of AQL, which already have been implemented in graphANNIS, have been included in the workload. There are several rarely used binary and unary operators in AQL which are not yet implemented. All features of AQL that have been described in Section 3.2 are implemented in graphANNIS and included in the workload. As the graphANNIS implementation gets more comprehensive in the future, the more queries can be included. Also, only queries that use a single corpus have been included because querying multiple corpora in one query is not implemented yet.

Duplicates Duplicated queries have been removed from the workload such that each query is included only once. Queries are assumed to be duplicates of each other if their JSON representation returned by the AQL parser is identical. Thus, only syntactic identical queries are considered to be duplicates, queries that yield the same results are not eliminated.

All in all, 4354 queries using 18 different corpora have been included in the workload (see Table 7.1 for a complete list of the corpora). Figure 7.1 shows how many queries have been excluded for which reason. Before filtering, 6372 single-corpus queries have been collected for the 18 selected corpora. Of these queries, 805 (about 12.63%) have been excluded because they use query language features which are not available in graphANNIS yet. This shows that graphANNIS already supports a relevant portion of AQL. The duplicate elimination filtered out additional 1114 queries (20.01%). Additional manual filtering was necessary for 99 queries that either triggered bugs in the legacy relANNIS implementation or used regular expressions which are not supported by the external RE2 library used by graphANNIS. Figure 7.2 shows the distribution of the number of joins per query in the workload. More than 70% of the queries use none or only one join, which means that their join order is either fixed or not relevant. However, queries which have multiple joins are important to the execution time of the whole workload, which will be examined later in the experiments. For

	# queries
BeMaTaC_L1_2013-02.1 (Sauer 2013)	166
BeMaTaC_L2_2013-02.1 (Sauer 2013)	108
DDD-Tatian (Donhauser et al. 2015)	161
falkoEssayL1v2.3 (Reznicek et al. 2012)	73
falkoEssayL2v2.4 (Reznicek et al. 2012)	559
FalkoWHIGL2v2.1 (Hirschmann et al. 2008)	12
Fuerstinnenkorrespondenz1.1 (Lühr et al. 2015)	137
GUM (Zeldes 2016b)	1065
HIPKON (Coniglio et al. 2014)	42
KAJUK (Ágel and Hennig 2014)	40
kobaltL1v1.4 (Zinsmeister et al. 2012)	109
kobaltL2v1.4 (Zinsmeister et al. 2012)	284
Maerchenkorpus (Walter 2015)	95
Parlamentsreden_Deutscher_Bundestag (Odebrecht 2012)	683
pcc176 (Stede and A. Neumann 2014)	452
RIDGES_Herbology_Version4.1 (Odebrecht et al. 2017)	233
tiger2 (Brants et al. 2004)	37
TueBa-DZ.6.0 (Telljohann et al. 2009)	98

Table 7.1.: Listing of the corpora included in the workload and their number of queries. Some corpora are sub-corpora from the same corpus project, like `kobaltL1v1.4` and `kobaltL2v1.4`. These are counted as a separate corpus, even if they contain the same kind of annotations.

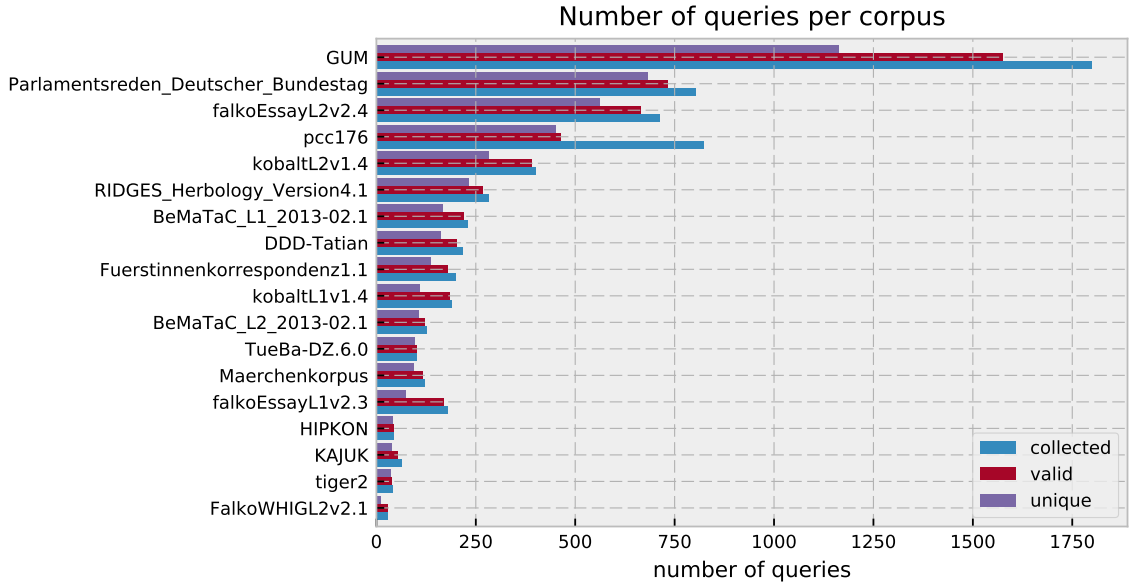
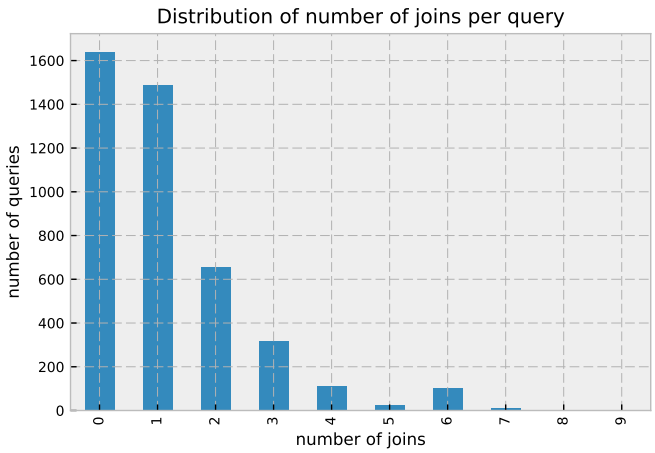


Figure 7.1.: This graphic shows the total number of collected queries per corpus and how many queries have been filtered out. Only queries that are valid AQL according to the graphANNIS parser have been marked as "valid". The next filtering step is the elimination of duplicates, which is shown by the "unique" bar.



	# joins
0	1641
1	1490
2	655
3	318
4	113
5	24
6	102
7	10
8	0
9	1

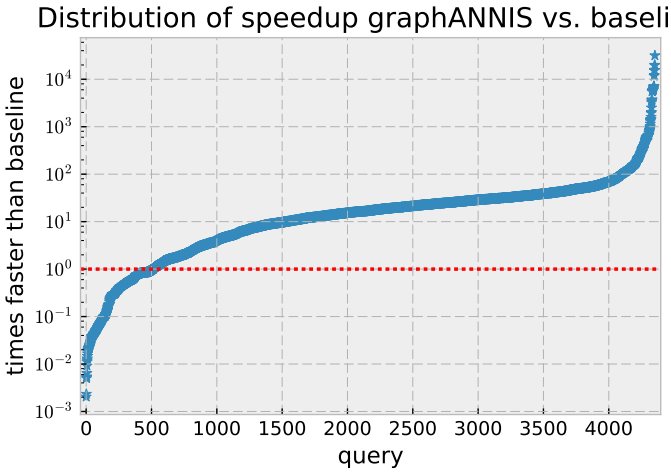
Figure 7.2.: Distribution of the number of joins per query for the complete workload of 4354 queries. The number of joins includes metadata search conditions: each metadata term will lead to an additional join of the first node with the `PartOfSubCorpus` operator, even if there is no explicit binary operator given in the query (see Section 5.6.6).

example, queries containing 4 joins describe quite complicated linguistic phenomena, and still, there are 250 queries in the workload (5.74%) with 4 joins or more.

The goal of following experiment is to compare the performance of graphANNIS with the existing solution relANNIS. Since relANNIS uses the disk-based PostgreSQL relational database and graphANNIS only uses main memory, PostgreSQL needed to be configured with enough cache to hold a complete corpus in main memory, so that is able to execute the benchmarks from this cache and to allow a fair comparison. For relANNIS, every query was executed 5 times initially to warm-up the cache.[4] Queries for each corpus where benchmarked separately and thus the cache only needed to hold the data for one corpus, not all of them. After the warm-up phase, each query of a corpus was executed 5 times in random order. The mean execution time of the 5 runs was then used as execution time for the single query. RelANNIS uses a time-out to abort long-running queries, which is normally set to one minute. Since the benchmark included queries that run much longer than this, the time-out was extended to 100 minutes. Recent versions of both relANNIS (3.5.0-preview4) and PostgreSQL (9.6.3) where used. PostgreSQL was configured with a shared buffer of 8 GB and a per-process working memory of 128 MB.[5] Only one query was executed at the same time and the optional parallelization features of PostgreSQL 9.6 where not enabled. GraphANNIS benchmarks where executed similar, with 5 runs per query and using the mean of these runs as execution time. The version of graphANNIS used for

[4]The `iotop` system utility (`http://guichaz.free.fr/iotop/`, last accessed 2018-02-05) was used to monitor the benchmark on the largest corpus in terms of disk usage (`TueBa-DZ.6.0`). No disk-reads where recorded from PostgreSQL process after the warm-up phase.

[5]This is important because PostgreSQL needs to sort a large amount of data and the sort operation should be executed in main-memory, too.



(a) plot of the distribution

	times faster than baseline
0.10	58.71
0.20	37.73
0.30	29.09
0.40	22.61
0.50	16.93
0.60	12.27
0.70	7.61
0.80	3.03
0.90	0.83
1.00	0.00

(b) quantiles

Figure 7.3.: Distribution of the speedup for single queries. The dotted red line in the plot left marks the speedup factor of 1.0 (exact same execution time). Each query is one data point and the queries have been ordered by their speedup. As it can be seen from the quantiles of the distribution on the right, 80% of the queries are at least 3.03 times faster than the baseline implementation.

the benchmarks was 0.5.0. The Celero benchmark library[6] was used to organize and execute the benchmarks. Since graphANNIS was configured to preload all components of the corpus into main memory before executing the benchmark, no warm-up queries needed to be executed. All benchmarks have been executed on an isolated system with an Intel Core i7-4770HQ CPU (having a base frequency of 2.2GHz and a "Turbo" frequency of maximal 3.40 GHz) with 4 processing cores and hyper-threading enabled. The system has 12 GB main memory and is operated by an Ubuntu Linux 17.04 operating system. GraphANNIS was compiled with the GCC compiler version 6.3.0 and with the `O3` optimization level but was not tuned to the specific CPU model. The parallel execution was not enabled per default, except for the experiments in Section 7.2 where the impact of parallelization is examined specifically.

Both the queries of benchmark and all measured results have been made publicly available at `http://doi.org/10.5281/zenodo.1161374` under a Creative Commons license. The corpora used in the benchmark, which are available under an open-access license, can be downloaded from `http://doi.org/10.5281/zenodo.1161383`. It is encouraged to compare the results of this work with other query systems and system configurations and it is planned to create a community project where new queries can be added to the workload by contributors.

7.1.2. Comparison of execution times

For comparing the complete workload, the mean execution times for each query have been added up, which leads to the following result:

	sum (in ms)
relANNIS (baseline)	2,617,695.60
graphANNIS	240,809.65

Thus, graphANNIS was able to execute the complete workload ~ 10 times faster than the baseline relANNIS implementation. A more detailed analysis of the speedup per single query is given in Figure 7.3. 80% of the queries are executed at least 3.03 times faster with graphANNIS compared to relANNIS.

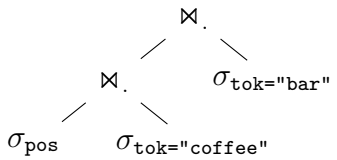
We analyze next, what kind of queries do not profit from graphANNIS and are faster when executed by relANNIS. The speedup of each query grouped by corpus or by the number of joins has been plotted in Figure 7.4. Both comparisons do not lead to conclusive results. While the absolute execution time of a query in graphANNIS is related to the number of joins (see Figure 7.5), the relative speedup per number of joins shows no clear tendency when compared to relANNIS. There are also differences in the speedup by corpus, but for all but one corpus queries are faster when executed by graphANNIS. This problematic corpus `falkoEssayL2v2.4` contains very similar annotations as another non-problematic corpus in the workload (`falkoEssayL1v2.3`) and the structure of the annotations cannot explain the difference.

In order to find problematic query classes, a more deep analysis based on the syntax of the queries itself has been performed. For each query, the parsed syntax tree (but not the actual result size and cost estimations) was used to decide to which class this query is assigned to. Then, each query was plotted with both its execution time for graphANNIS and relANNIS. See Figure 7.6 for the resulting plot. From this plot, clusters of the classes that have worse execution times can be identified. In this specific workload, there are the following two problematic classes:

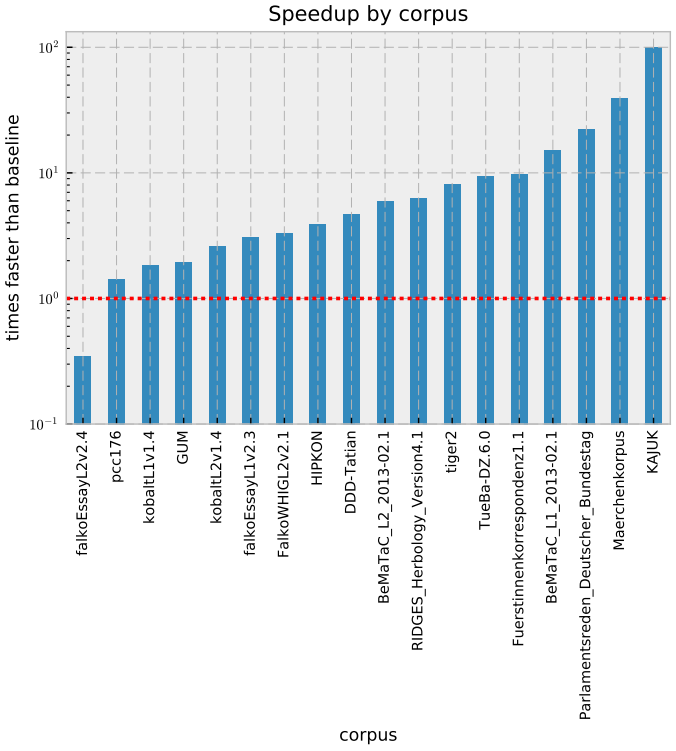
Root node without value and without edge annotation This includes queries having a query root node (a node which is only on the LHS of all operators and thus will probably be used as root iterator in an execution plan) which has no value definition and is not part of any operator with an edge annotation. An example is the AQL query

```
pos & tok="coffee" & tok="bar" & #1 . #2 & #2 . #3
```

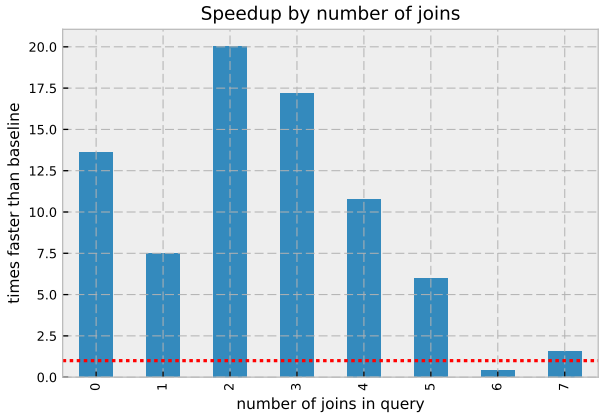
which would result in the execution plan



[6] `https://github.com/DigitalInBlue/Celero` (last accessed 2017-12-08)



(a)



(b)

Figure 7.4.: Speedup for different corpora (a) and the number of joins (b). The dotted red line marks the speedup factor of 1.0 (exact same execution time). The speedup has been calculated by using the sum of execution times for the complete workload of each group.

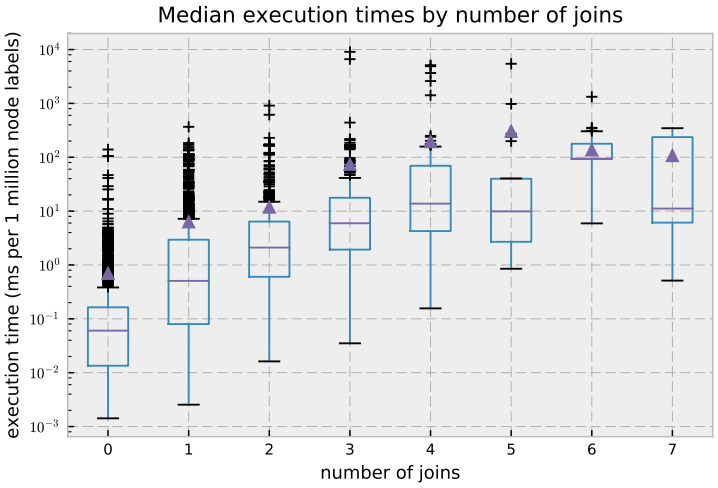


Figure 7.5.: Box plot of the normalized execution times by the number of joins. The execution times have been normalized to the size of the corpus (measured in the number of node labels) to make them comparable.

This query has a search for the annotation name `pos` as an initial iterator of the plan. Using only the annotation name has a lower selectivity than having an additional filter criterion like an annotation value or an outgoing edge with a specific annotation. An `IndexJoin` implementation would be used in such a left-deep join tree and it will search reachable nodes for each of the many results of the LHS. Queries of this form are often used when a frequency analysis is performed on the results. The exported results need to include an annotation node which is part of the linguistic phenomena, but its value should not be restricted. In the frequency analysis on this export, the different variations of the annotation value for the non-restricted node can be compared to each other.

Single node without value and with meta annotation This class is similar to the previous case with the difference that queries of this class only have one annotation node search (again without a filter for the value which would increase the selectivity) and a join with a meta condition. An example query would be `pos & meta::name="John Doe"`. Since edges of the `PART_OF_SUBCORPUS` component connect the annotation nodes with the corpus nodes and not the other way around, an `IndexJoin` operator would need to use the less selective `pos` annotation search as root iterator.

If we re-analyze the speedup for different corpora with all 298 queries that belong to one of these problematic classes excluded, all corpora including “falkoEssayL2v2.4” are executed faster in graphANNIS than in relANNIS (see Figure 7.7). Executing the workload of non-problematic queries for all corpora with graphANNIS is $\sim$ 19 times faster than executing the same queries with relANNIS. Also, the distribution of the speedup shifts in favor to graphANNIS, which can be seen by comparing Figure 7.3b with Figure 7.7b: The latter one shows that 90% of queries in the non-problematic workload are at least 1.57 times faster.

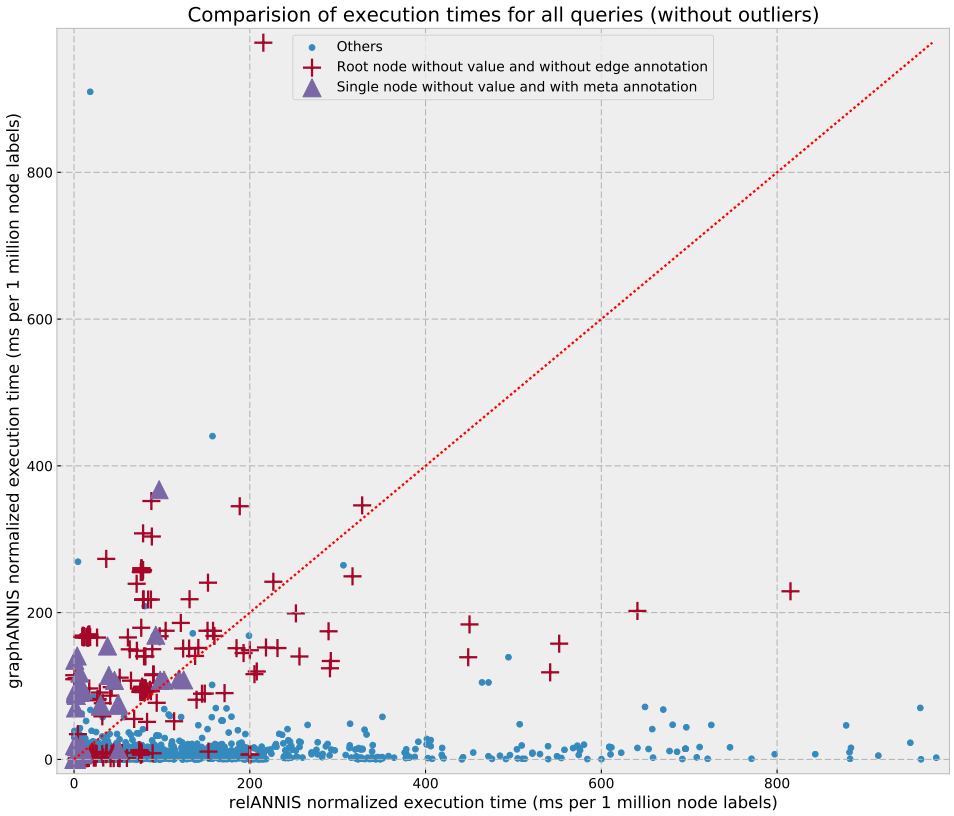
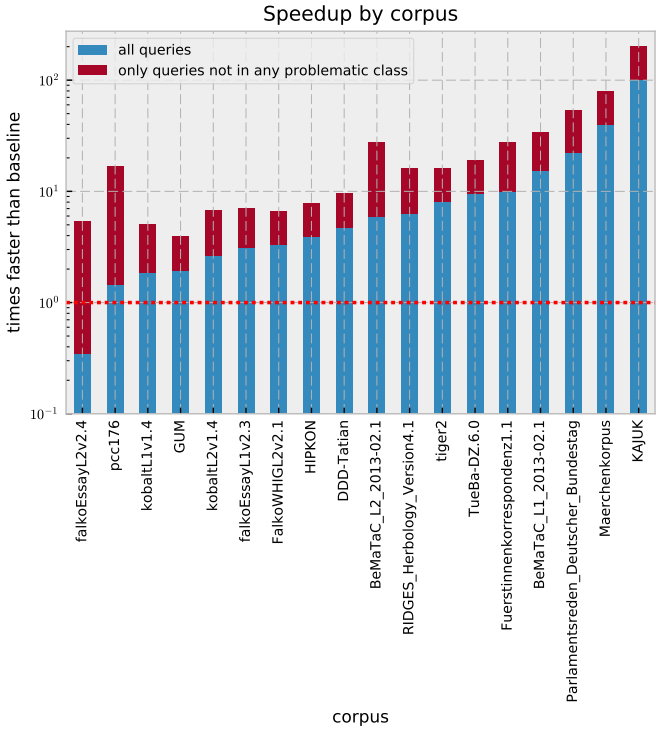


Figure 7.6.: Comparison of execution times for all queries. Outliers (the slowest 0.2% of the queries) have been removed for better visibility. All data points are plotted with both their normalized execution time for relANNIS (x-axis) and graphANNIS (y-axis). The execution time is normalized by the number of node labels of the corpus and is the same as in Figure 7.5. Each query that is plotted below the dotted red line is executed faster in graphANNIS than in relANNIS and vice versa. Additionally, each query is assigned to one of three categories and marked in the plot. It can be seen that most queries that are slower in graphANNIS compared to the baseline are either in the "Root node without value and without edge annotation" or "Single node without value and with meta annotation" category. This is not an "if and only if" connection: Queries that belong to these two categories can still be faster compared to the baseline. But if a query is known to be slower, it is probably in one of the identified problematic classes.



(a) speedup by corpus

	times faster than baseline
0.10	62.20
0.20	38.89
0.30	30.13
0.40	23.88
0.50	18.62
0.60	13.89
0.70	9.37
0.80	4.62
0.90	1.57
1.00	0.00

(b) quantiles

Figure 7.7.: Effect of the two identified problematic classes of AQL queries. In (a) the speedup for different corpora with problematic query classes excluded is shown. The original speedup for all queries is also included for reference. For all corpora, removing the queries from the two problem classes results in a better performance of graphANNIS compared to relANNIS. Also, the quantiles (b) show that the overall performance is better if the problematic query classes are excluded (the original quantiles are given in Figure 7.3b).

The problem in both these classes is an LHS with potentially low selectivity.[7] Since some operators are not commutative, the query optimization will not switch the operands and use the `IndexJoin` implementation for queries with only one join. For queries with multiple joins, the join order might be selected in a way that the problematic join is implemented as `NestedLoopJoin`, which can be slow, too. In contrast to graphANNIS, the relANNIS implementation uses arithmetic comparison operators on order values to find reachable nodes and can switch the operands of these comparison orders when needed. For graphANNIS, it would be desirable if there would be an inverse operator available for all non-commutative operators. For example, for the `Precedence` operator, this inverse operator would find all RHS nodes that come before a given LHS in the token stream. Such inverse operators might need access to the inverse edges of a graph storage. Depending on the graph storage implementation, these inverse edges are already stored and for other implementations, this feature could be added. Adding inverses edges to the graph storages would lead to a larger memory consumption, but this trade-off seems justifiable given that this would enhance the execution speed of almost all problematic queries in the workload.

7.1.3. Comparison of output size and cost estimation

Estimating the output size of an operation is an important aspect of the optimizing process because the cost estimation determines which joins are executed in which order. To test the accuracy of this estimation, the execution plans for both graphANNIS and relANNIS have been collected for all queries that are included in the workload and the estimated output sizes have been extracted. The queries were filtered to only include queries with at least one result, which makes calculating the relative estimation accuracy easier.[8]

Two different kinds of estimations are implemented in graphANNIS: Estimating the output size of an annotation search and estimating the result of a join operation. The estimation of annotation search is the base for the join estimation and thus highly influential in the overall accuracy. In order to measure the difference between the estimation and actual number of results, the error ratio

$$\frac{|n_{actual} - n_{estimated}| + n_{actual}}{n_{actual}} \tag{7.1}$$

is used which allows treating both under- and over-estimations in the same way. For instance, if the actual output size is 100, both the estimations of 120 and 80 would result in an error ratio of 1.2. Figure 7.8 shows what margin of error can be expected from both graphANNIS and relANNIS, by plotting how many percents of queries do not exceed a given maximum error ratio. In comparison to relANNIS, graphANNIS

[7] In theory an annotation name could be rare, too. But since the categorization is only based on the parsed query and has no knowledge of the statistics of the corpus, the simplified assumption that queries for annotation names are less selective than the ones for an annotation value seems appropriate.

[8] Also, queries without an actual result can be faster to execute, if the query system can detect that they do not produce any results and do not actually need processing. Thus, this simplification can be justified with the larger relevance of queries with at least one result.

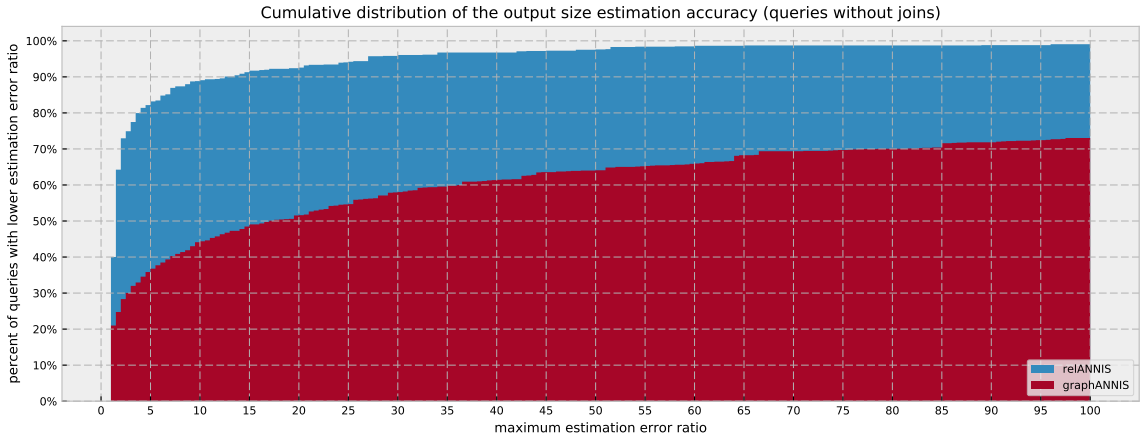


Figure 7.8.: Cumulative distribution of the output size estimation accuracy for queries without a join. The x-axis depicts a given maximum estimation error ratio and the y-axis shows for how many percent of queries in the workload this error ratio is not exceeded by the predictions of either graphANNIS or relANNIS.

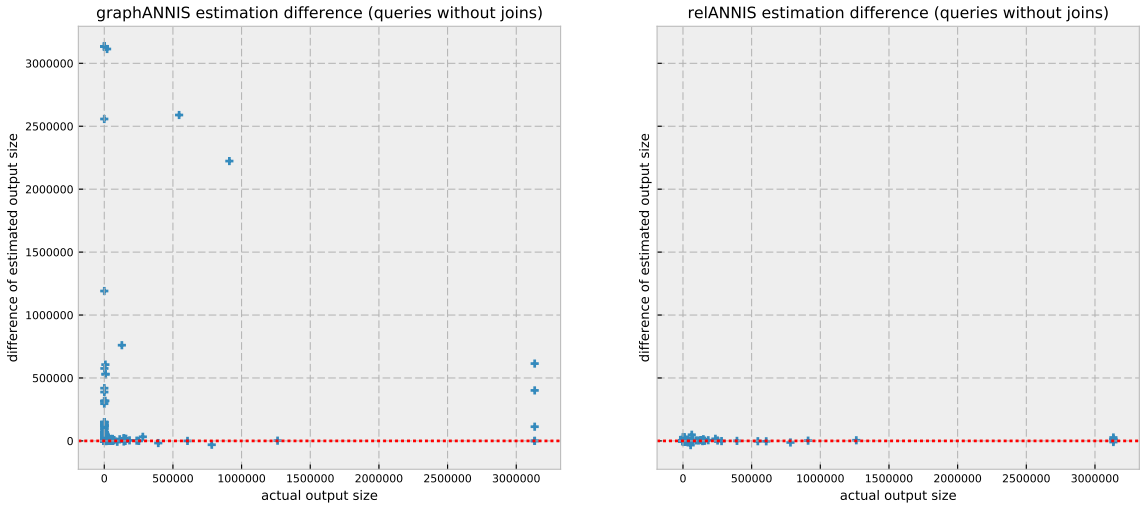


Figure 7.9.: Estimation difference for annotation searches. This does not show the error ratio, but the relative (possible negative) difference in the output size estimation.

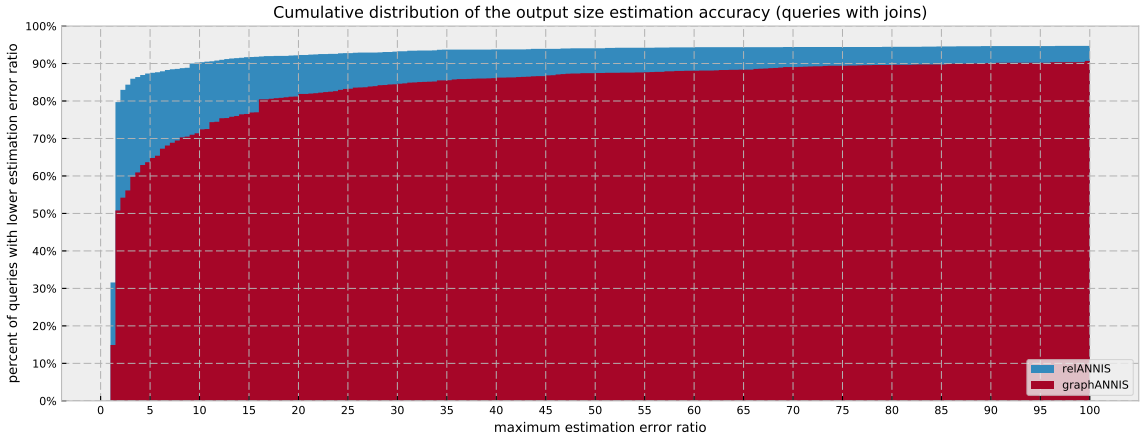


Figure 7.10.: Cumulative distribution of the output size estimation accuracy for queries with at least one join. The x-axis depicts a given maximum estimation error ratio and the y-axis shows for how many percents of queries in the workload this error ratio is not exceeded.

predicts single node annotations poorly. For each given maximum error rate, the number of queries for which the prediction of relANNIS is inside this error rate is larger than the ones where the prediction of graphANNIS is inside the same allowed margin of error. For instance, about 50% of queries in graphANNIS do not exceed the error ratio of 20 (the estimation is 20 times larger than the actual size), while about 90% percent of the queries in relANNIS do not exceed the same error ratio limitation. This shows how good recent versions of PostgreSQL are in predicting estimations even in complex situations like regular expressions. As it can be seen in Figure 7.9, graphANNIS often largely over-estimates the number of results for node annotation searches, while PostgreSQL shows a much more balanced prediction behavior. If this over-estimation is consistent, it might not influence the join order optimization much, but the node annotation search estimation is still an area where graphANNIS clearly needs improvements.

The estimation of the results for joins seems better, as Figure 7.10 suggests. It shows the expected margin of error for all queries that have at least one join. Although this estimation incorporates the problematic results of the node annotation estimations, graphANNIS calculates much better estimations compared to the queries without a join. For instance, now 80% of the queries have a maximum error rate of 20 (compared to 50% for queries without a join). PostgreSQL also performs better for the queries with a join, but the difference is smaller compared to the annotation search queries.

If the output size estimation of PostgreSQL is better for the workload, it raises the question of why PostgreSQL performs so much worse when the execution times are compared. Also, previous experience in examining problematic queries submitted by users of relANNIS showed that PostgreSQL sometimes under-estimated intermediate size results and chooses inadequate plans given the number of intermediate results. Figure 7.11 compares the cost estimation of the systems for queries with joins with their measured execution times. GraphANNIS and relANNIS show similar plots and both also have a similar Spearman's rank correlation coefficient (Steland 2016, p. 59):

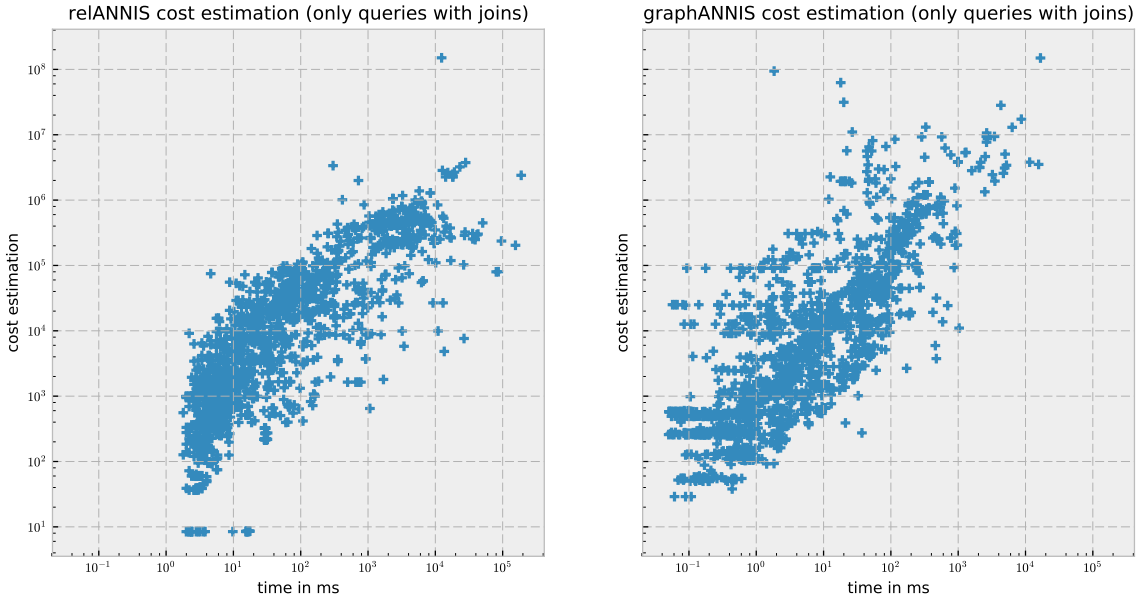


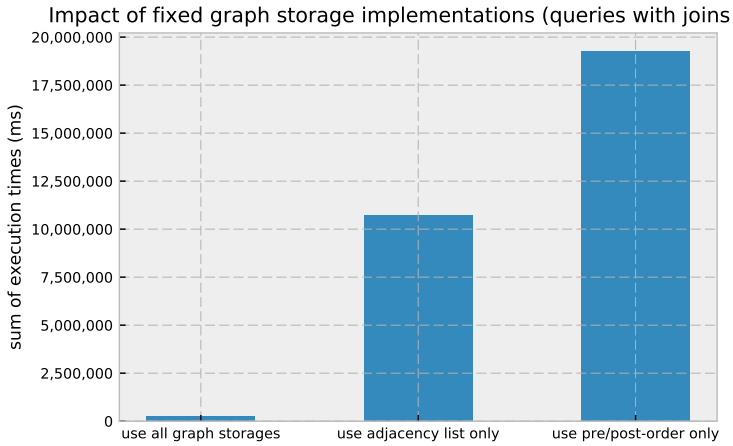
Figure 7.11.: Comparison of the cost estimation of graphANNIS and relANNIS. This plot shows the internal cost estimation of both systems in relation to the actual execution time of the query in the particular system

$\rho = 0.81$ for graphANNIS and $\rho = 0.82$ for relANNIS. This means their relation between estimated cost and execution time can be described by a monotone function with similar accuracy. Thus, despite the huge differences in the quality of the node annotation search prediction, the simple cost estimation model of graphANNIS still results in comparable plan quality. Given that graphANNIS has fewer possibilities to implement the different operations in the execution plan, the search space for the optimal plan is also smaller. As long as the cost estimation is still able to distinguish a better plan from a worse one, a smaller search space has the advantage that it can be explored more extensively in a smaller amount of time than it would be possible with a larger one.

7.2. Impact of optimization and parallelization

In the previous section, the relational database implementation and graphANNIS have been compared and it was found that the execution time of graphANNIS for a realistic workload is about 10 times faster than the one of relANNIS. This chapter evaluates which optimization strategies have which impact on the result. The same workload and benchmark results as in the previous section are used. They have been extended with benchmark results of various new configurations of graphANNIS.

One major contribution of graphANNIS is the provision and automatic selection of specialized graph storages for different kind of edge components and their integration into a single system. It is argued that the possibility of using these differently optimized graph storages in the same system improves its performance. GraphANNIS has already been compared to relANNIS, a system that only uses one representation of graph



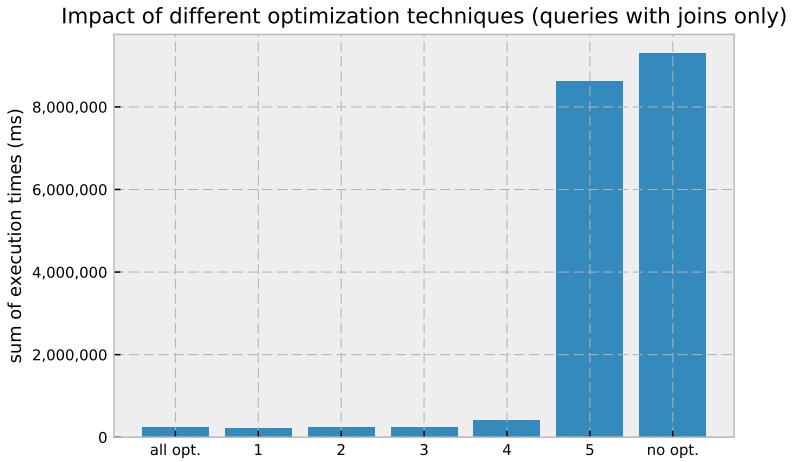
	sum (in ms)
use all graph storages	229,196.62
use adjacency list only	10,743,124.10
use pre-/post-order only	19,253,429.60

Figure 7.12.: Impact of fixed graph storage implementations on queries with at least one join. The baseline is the default configuration, where all graph storage implementations are used. In the other configurations, only one graph storage implementation was used for all components.

components (the pre-/post-order encoding). It could be argued, that pre-/post-order encoding is just an inferior representation for linguistic annotation graphs and that using another representation like adjacency lists and graph traversal would be faster. This was tested by benchmarking different configurations of graphANNIS, where only one single graph storage implementation was used.[9] All queries of the workload with at least one join have been included in this test. The sum of execution times for each configuration can be seen in Figure 7.12. Using only one implementation leads to a drastic loss in performance: The workload is executed either ~ 47 times or ~ 84 times slower respectively. This is a clear indication that choosing graph storage implementations based on the structure of the annotation graph is a worthwhile optimization and that the combination of different techniques to implement graph reachability queries is an important factor in the overall system performance of graphANNIS.

Using different strategies for representing the annotation graph is not a technique that can only be used in a specialized query engine like graphANNIS. For example, in a relational database, different table definitions could be used for different types of graphs. Queries on all graph storages of a database, disregarding the type, could be performed with the help of views, stored procedures or application specific logic when the SQL is generated. However, the complexity of the existing approaches to

[9]Since the `LinearStorage` cannot represent DAGs, it has not been used as a test configuration. Also, the `LEFT_TOKEN` and `RIGHT_TOKEN` components used the adjacency list implementation in all configurations since these components are cyclic and cannot be represented in pre-/post-order encoding.



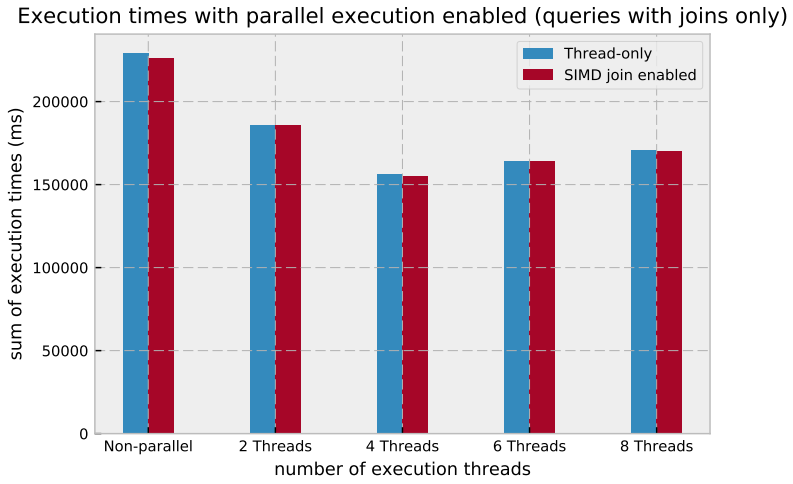
		sum (in ms)
	all optimizations	229,196.62
1	no exhaustive join order permutation (only heuristic)	227,437.80
2	no operand order	236,100.41
3	no unbound regex	236,150.98
4	no node by edge anno.	408,325.08
5	no join order opt.	8,625,239.43
	no optimization at all	9,290,923.66

Figure 7.13.: Impact of different optimization techniques on queries with at least one join. The lower baseline is the default configuration, where all optimizations are enabled and the configuration with no optimizations at all is the upper limit. In the other configurations (numbered from 1 to 5), all optimizations have been enabled, except the one listed in the table.

implement graph reachability queries in relational databases (see Section 2.2.3), could be multiplied when several of these are used in the same system.

GraphANNIS implements several techniques to optimize the execution plan of a query and these techniques have been evaluated as well. To test what kind of impact they have on the overall performance, two extreme baseline configurations have been defined. One is the default configuration with all optimizations enabled and the other one is a configuration where no plan optimization is enabled at all. For each optimization technique, a configuration has been added where all optimizations have been enabled except the technique which influence should be measured. Thus, if the execution performance drops when disabling a technique, it must be an important factor in the system performance.

The results for the different configurations can be seen in Figure 7.13. Using only the heuristic optimizer for determining the join order does not influence the execution time negatively. It seems like the heuristic optimizer is always able to find an equally good join order compared to the exhaustive plan space exploration. Testing all permutations even leads to a slightly higher execution time because of the increased computational cost of planning the query. Switching the operand order and replacing unbound



	Thread-only (sum in ms)	SIMD join enabled (sum in ms)
Non-parallel	229,196.62	226,236.49
2 Threads	186,054.51	185,923.93
4 Threads	156,485.25	155,241.36
6 Threads	163,858.23	164,132.16
8 Threads	170,529.74	170,235.26

Figure 7.14.: Parallel join execution times for queries with at least one join. The system on which the benchmark has been performed has only 4 cores and thus for executing the configuration with 6 and 8 threads, hyper-threading was used.

regular expression seems not to have any major effect on the workload. The exchange of non-selective node annotation queries with more selective queries on nodes having a specific outgoing edge has much more influence. Deactivating this optimization almost doubles the execution time. The by far greatest influence is the join order optimization. Without this optimization, the execution time for the workload is increased by the factor 37.63. Even with an output size estimation worse than relANNIS and a limited set of implementations for different execution plan operations, it is still crucial to have these basic statistics to find a good join order. It would be interesting to see if an improvement of the output size estimation would lead to better execution times, or if the current cost estimation is already good enough to distinguish the best join order from the other ones.

Another optimization was the implementation of parallel joins. These have been benchmarked separately and the result can be seen in Figure 7.14. The results are not surprising. Parallelizing joins with 2 thread decreases the execution time of the workload by 18.82% and adding another 2 threads results in a decrease of 31.72% compared to the non-threaded version. Since the system on which the benchmarks where executed only has 4 cores, adding more threads means hyper-threading was used. In this workload, hyper-threading does not help to increase the speed but slows down the query execution time. This could be caused by the overhead introduced by more threads that need more synchronization, which might not be compensated

enough by the limited additional parallelization provided by hyper-threading. In order to investigate this hypothesis, the application was profiled with the query

```
infstat="new" & cat > tok & #1 _=_ #2 & tok & #4 ->dep[func="pobj"] #3
```

on the GUM corpus in three configurations:

- with no threading activated,
- with 4 threads (maximal number of physical cores on the test computer), and
- with 8 threads (hyper-threading with two threads per physical core).

Query plans will have a `NestedLoopJoin` as root execution node for this query and this join was the hot spot in the overall execution. In case of the non-threaded version, $\sim 3\%$ of the total CPU cycles were spent fetching the next pair of match candidates inside the loop of the join. When 4 threads where used, the whole nextPairSynchronized function of Algorithm 6.2 needed $\sim 47\%$ of all cycles. This overhead is caused by the locking and unlocking, that alone needed $\sim 43\%$ of all CPU cycles needed to execute the query. For 8 threads and activated hyper-threading, the overhead introduced by locking in this single function was further increased and needed about $\sim 53\%$ of all cycles. This exemplary analysis shows that the overhead for thread-based parallelization in the current implementation has a large influence and should be a target for future optimizations.

All configurations also have been executed with the SIMD join enabled. In case of the "Non-parallel" configuration, this meant that the `SIMDIndexJoin` was used whenever possible and the non-threaded `IndexJoin` or `NestedLoopJoin` where used otherwise. For the other configurations, the `SIMDIndexJoin` was used for all compatible operators that have not already been implemented with a threaded join. As it can be seen in Figure 7.14 the impact of the `SIMDIndexJoin` is minimal. This is not surprising given the limitations in which cases it can be used at all and the amount of work it does perform in parallel. SIMD has been used to build more efficient index structures like in Sprenger et al. (2017). Those implementations are carefully designed to not only use SIMD instructions but also consider other important aspects like proper cache line usage. Using such an index structure in graphANNIS might provide more performance gain than applying SIMD to joins.

7.3. Main memory usage

GraphANNIS aims to support corpora as large as possible. Since it is a main-memory query-system, the memory consumption of a corpus relative to its size is an important factor. Because the original relANNIS implementation runs on both server and desktop systems, it is important that graphANNIS can at least handle the same corpora on the same hardware. Additionally, graphANNIS should scale well to large corpora: Support for larger corpora should only be an issue of adding main memory to the system. As it was discussed in Section 3.3.3, this used to be a principal problem with relANNIS.

	memory (MB)	# node labels	# strings	# tokens
TueBa-DZ.6.0	3,505.81	20,412,959	4,758,713	975,836
falkoEssayL2v2.4	1,482.39	11,795,251	2,841,871	144,619
Parlamentsreden_Deutscher_Bundestag	1,443.45	18,805,438	3,236,263	3,134,192
tiger2	877.22	7,719,707	1,371,806	888,552
Fuerstinnenkorrespondenz1.1	869.82	6,737,207	1,741,553	262,465
RIDGES_Herbology_Version4.1	752.55	5,608,666	1,332,469	154,267
FalkoWHIGL2v2.1	633.14	6,677,346	1,091,275	130,949
falkoEssayL1v2.3	591.85	4,766,271	1,134,571	70,615
kobaltL2v1.4	504.02	4,023,107	989,330	33,368
kobaltL1v1.4	199.72	1,573,751	389,393	12,984
KAJUK	156.41	1,204,567	318,401	119,420
Maerchenkorpus	150.15	1,776,987	318,965	295,880
HIPKON	138.79	970,430	268,797	109,045
GUM	135.82	957,216	165,676	64,005
DDD-Tatian	104.02	1,134,381	199,876	54,677
pcc176	51.31	381,859	69,099	33,298
BeMaTaC_L2_2013-02.1	49.57	344,884	99,846	12,517
BeMaTaC_L1_2013-02.1	46.23	317,767	90,553	11,187

Table 7.2.: Size of the different corpora. This includes the measured memory consumption of a corpus when fully loaded into main memory, the number of node labels, the number of unique strings and the number of tokens. The table is sorted by the used main memory.

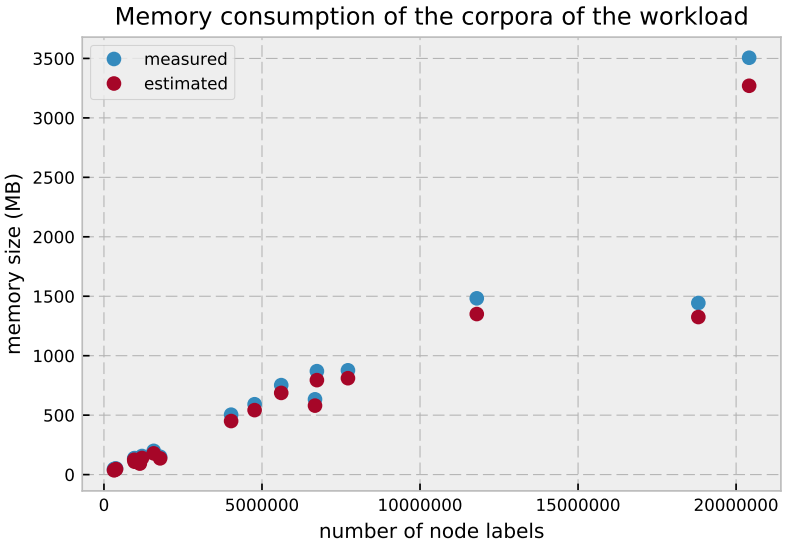


Figure 7.15.: Memory consumption of each corpus of the workload compared to its number of node labels. There are two sizes reported for each corpus. One is the memory consumption measured by the operating system. Since this functionality is not available on all supported operating systems, graphANNIS estimates the used main memory by inspecting the size of its data structures, too.

To examine the feasibility of using differently sized corpora in graphANNIS, the resource usage of all corpora that have been used in the benchmark workload have been measured. For simple text corpora, the number of tokens is often used as a reference of their size. But comparing the size of multi-layer corpora with each other is not that straightforward because they can add multiple annotation layers. This increases the number of nodes of the corpus without increasing the number of tokens. Thus, a corpus with more annotation layers than another one can be more complex even if it has fewer tokens. In addition, annotations are not only expressed as nodes, but also as edges between nodes which makes the comparison even more difficult. Table 7.2 shows all corpora in the benchmark workload with their number of node labels, the number of distinct strings and the number of tokens. It also lists the used main memory size when all components have been loaded. The largest corpus in terms of main memory size is the `TuebaDZ6.0` corpus, which needs about 3.5 GB of main memory. This corpus is annotated with syntax annotations with many edges and additional nodes but it only has 975,836 tokens. In comparison, the `Parlamentsreden_Deutscher_Bundestag` corpus, which has more than 3 million tokens, needs only about 1,5 GB main memory. Since the latter corpus only contains token-wise annotations, the number of node labels is smaller and memory usage is much lower. Even for the `tiger2` corpus, which is similarly annotated as `TuebaDZ6.0` and also has a similar tokens size, the memory usage is much less because it has fewer node annotation labels. The plot in Figure 7.15 shows the relation between the number of node labels and the used memory size. All in all, the largest corpus in the workload needs about 3.5 GB of main memory and smallest one 46 MB. Currently, notebook systems with 8 GB of RAM are common and reasonable priced server systems can be purchased with 128 GB of RAM. This means that even a corpus with more than 30 times the size of `TuebaDZ6.0` or 140 times the size of the `tiger2` is expected to run without problems on current mainstream server hardware.

8. Conclusion and future work

The previous chapters presented graphANNIS, a main-memory query system for linguistically annotated corpora. In this chapter, the main contributions are discussed, and possible directions for advancements and further studies are described.

8.1. Relevance to corpus linguistics

GraphANNIS implements large parts of the existing query language AQL, supports all corpora of the legacy ANNIS system and can be used as a drop-in replacement of the existing back-end. From its functionality, it is therefore at least as relevant for the community as the existing system, as soon as missing functions like additional operators are added. This is an engineering effort that seems feasible, given the current architecture of graphANNIS and the already existing features.

The legacy ANNIS system has been used to conduct various studies and has been thought to researchers in tutorial sessions and books like Kübler and Zinsmeister (2014). While it has very good support for corpora with different types of annotations and a powerful query language, its problems are non-functional, especially scalability and faster response times for certain queries and corpora. ANNIS was always assumed to work well on "small" corpora, but unusable for "large" ones (Kübler and Zinsmeister 2014, p. 270). What "large" means highly depends on the context; for instance corpora that are completely automatically generated from crawling the Internet and adding unsupervised annotations like the COW corpora (Schäfer 2015) can include several billions of tokens. For certain research questions, these automatic annotations are not enough (for example the example from Chapter 1 needs manual syntax annotations), and for some historical languages, the number of preserved texts is also very small. Corpora that need manual annotation tend to be much smaller, but are still very useful for research. These "small" corpora can grow over time when new annotation layers and texts are added, especially if they are collaborative efforts of different researchers that use the same texts for answering different research questions. Research data repositories like LAUDATIO (Krause et al. 2014) foster the re-usage and extension of such corpora. For large corpora that only use limited types of annotation, using the existing CWB tools has been a viable approach, but for "growing" corpora, where new and maybe even currently unknown types of annotations are added, this is not possible.

For such types of multi-layer corpora, flexible graph-based data models like Salt have been successfully used to represent the data. AQL is a query language based on annotation graphs. But although its legacy relANNIS implementation was able to represent the corpora in a flexible way, it was not always querying these graphs efficiently. Like the more specialized query systems (like the ones described in Section 2.3), it

was optimized for one type of annotations (in this case tree-like ones). Using graphs as abstraction adds flexibility, but it shadows the actual structures and regularities of the underlying annotation layers. Each annotation layer has its own regularities and systems like relANNIS, which partition the corpora only into documents to reduce the search-space, cannot make use of these regularities since each document contains all types of annotation layers. GraphANNIS is able to exploit the different properties of these structures, by partitioning the corpora into edge components. This reduces the search space and allows optimized implementations for known types of graphs. The evaluation showed that this kind of specialization contributed considerably to the overall performance improvement of the graphANNIS system compared to its predecessor relANNIS. Since graphANNIS adopts the flexible query language from relANNIS but avoids many of its drawbacks implementation-wise, it can be used as a valuable tool in corpus linguistic research, among the other existing solutions.

8.2. Comparison to existing solutions

In Chapter 2, several other existing approaches and technologies for querying linguistic corpora have been described. This section discusses the advantages and disadvantages of different aspects of graphANNIS compared to these existing solutions.

8.2.1. Using commercial off-the-shelf systems

Changing the implementation of ANNIS from using a relational database to implementing the search functionality all by itself was a paradigm shift from relying on a commercial off-the-shelf (COTS) system to a custom implementation. Using a COTS system like PostgreSQL, Neo4j or Apache Lucene (see Section 2.3 for approaches that use them for linguistic corpus query systems) has the advantage of needing fewer development resources and allows profiting of ongoing developments in the database and index technology. For instance, in a previous benchmark of graphANNIS with a similar workload, but against an older version of both PostgreSQL and relANNIS, graphANNIS was about 40 times faster for the complete workload (Krause et al. 2016) compared to "only" 10 times faster in the evaluation against newer PostgreSQL and relANNIS versions as described in Section 7.1. Further developments like parallel query execution in PostgreSQL 10 will probably lead to further improvements. Choosing a COTS system still has the downside of limited possibilities to make use of the specific characteristics of the linguistic annotations and the read-only query scenario. For instance, mapping an annotation graph to a relational database will always pose challenges similar to those described in Section 3.3.3.

However, some of these problems can be avoided with generic COTS solutions, too. In Rosenfeld (2012), the main memory column-store MonetDB[1] was used to execute AQL queries. Its column-oriented approach allowed the usage of the original normalized relational schema of relANNIS (described in Section 3.3.2) and avoided duplication caused by using a pre-joined materialized view. Still, the initial port of ANNIS from PostgreSQL to MonetDB with minimal changes to the generated

[1] `https://www.monetdb.org/` (last accessed 2018-01-26)

SQL showed a degraded performance on a benchmark based on the `tiger2` corpus. Execution speed could be improved by the factor 23 compared to the initial port by changing the generated SQL considerable and by improving the implementation of MonetDB itself (Rosenfeld 2012, p. 81). The most influential change in the SQL was that parts of the results have been pushed down in the execution plan manually with help the of Common Table Expressions (CTEs), effectively overriding the query planners decisions and optimizing the query plan from the application (Rosenfeld 2012, pp. 52 ff.). In comparison to PostgreSQL, the improved version of MonetDB was able to execute the queries on the `tiger2` corpus in less than a third of the time on a server system and about ten times faster on a desktop system (Rosenfeld 2012, p. 79). While the results of this benchmark support the assumption, that a main memory-based system can also query larger corpora like `tiger2`, it also shows that extensive adjustments to the COTS system were needed to make use of the specifics of the domain of linguistic annotation graphs. Also, the benchmarks were performed on a corpus with syntax trees only, a structure that is very well supported by the pre-/post-order encoding used to encode the graph. But it does not solve the problem of mapping different types of annotation graphs efficiently inside the same database.

Using an actual graph database could solve problems introduced by mapping the data to a relational database, but existing solutions like Neo4j do not include the flexibility to adapt to different graph structures. In the case of Neo4j, finding reachable nodes is implemented by graph traversal[2], which is an unsuitable approach for several types of annotation graphs like those with long paths (see Section 7.2 for an experiment where only graph traversal was used). While it might be possible to work around such limitations, such custom solutions will also need to be adjusted to changes in the underlying platform and thus need continued development effort. Also, a specialized implementation can be a testbed for advanced technologies under active research. Since the domain is more limited, but there is still a non-artificial use-case scenario, a linguistic query system could give valuable insight into techniques like graph indexes such as Grail (Yildirim et al. 2010) or Ferrari (Seufert et al. 2013) in future studies.

8.2.2. GraphANNIS as open-source community project

The downsides of the increased development effort, when not relying on a COTS solution, could be reduced by developing such a tool as an open-source community project. ANNIS is already part of a set of tools around the common data model Salt named “corpus-tools.org”, which are developed in such a community effort (Druskat et al. 2016). Its general data model allows using ANNIS for a broad range of corpora, which allows it to represent corpora from different fields of linguistics and even other fields that use annotations on texts, like for example literary science. Such broad applicability leads to a larger user-base compared to tools which are more restricted, like the ones that only support one type of annotation.

In order to actually allow a distributed development, the entry level for contributing should be as low as possible. GraphANNIS is written in C++ (the other tools of

[2] In Robinson et al. (2013, p. 20), querying a graph is equated with graph traversal and the whole architecture of Neo4j is build around efficient traversal of graphs (Robinson et al. 2013, pp. 141 ff.).

"corpus-tools.org" are written in the Java programming language), and this can be a problem given the inapplicability of C++ as a programming language for beginners. It should be evaluated if other programming languages like Rust[3] can provide the same memory control and optimization possibilities as C++, but are safer to use for entry-level programmers who are common in the corpus linguistic community.

8.2.3. Embeddability of graphANNIS

Some COTS systems like PostgreSQL are based on a client-server infrastructure. Thus, users must either use a central web-based service from an infrastructure provider or install complex server software on their own computers. GraphANNIS does not require such server-systems and end-users will be able to install it more easily than relANNIS. Not using a server-software also allows integrating graphANNIS into other linguistic corpus tools more easily. For instance, by integrating a comprehensive query language into an annotation tool, it is possible to implement an agile corpus creation workflow where annotations are constantly checked for consistency and annotation schemes can be changed more easily (Voormann and Gut 2008). GraphANNIS has already been integrated into the Salt-based Atomic annotation editor (Druskat et al. 2014) as Java library to support such an agile workflow (Druskat et al. 2017). The query system CWB is equally embeddable and different customized front-ends make use of it as a query engine (Evert and Hardie 2011). Such front-ends could use graphANNIS or both systems in parallel in the future. They could allow accessing corpora in a way that is more specialized on a specific use case, instead of the general purpose approach of the current ANNIS web-interface. An example is the CALLIDUS project[4] which will study how to support the teaching of Latin in schools by providing access to Latin corpora to teachers and pupils. ANNIS will be used as a back-end, but the front-end will be highly customized with the possibility to adapt predefined AQL queries and generate exercises from the results, providing more simple access to the corpora than directly querying AQL.

8.2.4. Support for more complex and larger corpora

Another paradigm shift of graphANNIS was using exclusively main memory for accessing the data. All other corpus query systems described in Section 2.3 are disk-based (except the MonetDB based implementation of AQL). As it was shown in Section 7.3, even the largest corpora currently available in relANNIS should be supported on current desktop and notebook hardware. For larger corpora, central server systems which provide a web interface and a REST API could be used. Using main memory directly allows avoiding typical disk problems like caching and to concentrate the optimization efforts to other areas. Partitioning the corpora into edge components also allows applying simpler "caching" strategies like only loading the components relevant for a query into main memory. Other systems like KorAP have a more conservative disk-based design because they are explicitly designed to handle "very large corpora"

[3] `https://www.rust-lang.org/` (last accessed 2017-12-18)
[4] `https://www.projekte.hu-berlin.de/de/callidus` (last accessed 2017-12-18)

(Diewald and Margaretha 2016). In the case of KorAP, this includes the collection of corpora of the DeReKo project with in total more than 24 billion word tokens (Kupietz and Lüngen 2014). With larger main memory available for both server and desktop systems, graphANNIS is also intended to be a research platform where novel main memory approaches for graph-based linguistic query search can be tested.

All in all, as a corpus search tool that,

- supports multiple layers of annotations,
- allows representation and querying of different kinds of annotation,
- is graph-based,
- allows embedding in other tools, and
- is available as open-source and can be extended by the community

graphANNIS fills an important niche for corpus query systems. Each of these properties can be found in other query systems, but their combination is unique to graphANNIS.

8.3. Representativity of the workload

As part of this work, a benchmark workload of AQL queries has been created by collecting queries from publicly accessible server installations of ANNIS. This set of queries has been made available to the research community and can be used to compare different query systems with each other. One of the main filter criteria was the corpus they queried, but this can introduce a bias in the data. It was attempted to include corpora that represent different kinds of annotations, but it could be possible that users also need different kinds of annotations not included in the workload. Also, the corpora available on the server systems are only the ones that were available to the working groups that host these servers. Corpora that are not converted to ANNIS yet or are too large to be handled by the legacy implementation are not included. An additional bias can be introduced by the users of the servers that were used to collect the queries. While both servers are open to the public, their main user groups are the researchers associated with the particular research groups that host the servers. The queries they use, are influenced by the research questions they are working on, and researchers from the same working group might have similar research questions. By using two servers from two different working groups from different countries, this bias is hopefully reduced. The server at the Humboldt-Universität zu Berlin is also hosting corpora for other research projects, like the corpora from the Old German Reference Corpus project[5] or corpora that have been archived in the LAUDATIO repository[6]. If researchers of these projects also use this server, this should enhance the representativity of the queries. Since graphANNIS is intended to replace this legacy system and the comparison in Section 7.1 was between these two systems, the selection of the corpora and queries is not an issue for this specific comparison. But if

[5] `http://www.deutschdiachrondigital.de/` (last accessed 2018-01-26)
[6] `http://www.laudatio-repository.org` (last accessed 2018-01-26)

the benchmark workload is used with other systems and these systems target different types of corpora, this selection could be perceived as unfair. This could be solved by including queries from more corpora and other query systems. Alternatively, the benchmark workload can be seen as specific to the niche ANNIS attempts to fill and if other corpus systems claim to support the same niche, this workload can be a valuable tool to compare the different implementation approaches.

The large number of queries that could be collected (more than 4000 queries in total, see Section 7.1.1 for details) supports the overall representativity of the workload for the corpora that have been included.[7] Also, queries of a specific corpus have been only filtered out if they use an AQL feature not supported by graphANNIS yet. Outliers (queries that have been either very slow or very fast) remained in the workload. This leads to many queries that are not very complex and do not use many annotation nodes in the same query, which is counter-intuitive to the claim of ANNIS to support complex multi-layer corpora. These queries are still important, as users will iteratively make their queries more complex and supporting the "easy," but non-selective queries efficiently, is also part of this iterative process. If the system would fail to provide instant feedback, in this case, the overall perceived performance could suffer. Subsets of the workload containing only more complex queries can be created if necessary, for instance by including only queries with a certain number of joins. Another approach to identifying more "important" queries is by examining the persisted queries. This is a rather new feature of ANNIS, where users can create persistent links to queries or single matches of a query and share this links, for example in publications. Because the user has to explicitly trigger the creation of such a query link, intermediate queries of the interactive query creation process are less likely to be included.

8.4. Future work

This section gives an outlook on what future enhancements and studies could be performed with graphANNIS. First, additional possibilities of enhancing the query execution speed with more optimizations and parallelization are discussed. Then, functional enhancements are proposed.

8.4.1. Additional optimizations

Despite the better performance of graphANNIS compared to relANNIS, there are more optimizations that could be implemented or existing optimizations that could be enhanced. In order to get a better understanding of where the focus of solving these optimization problems should be, more experiments and measurements are needed.

[7] To the best of the author's knowledge, this is the largest set of realistic corpus queries, which has been made available. Publishing the query sets for published benchmarks is not a matter of course. Sometimes the query set is only vaguely described (for example in Rábara et al. (2017)), only the collection and selection process and the total number of queries is given (for example in Vanroy et al. (2017)), or a small manually designed set of queries is used (like in Meurer (2012) and Rosenfeld (2010)). A larger public data set is given in the appendix of Rosenfeld (2012). It contains 224 queries for the `tiger2` corpus, which have been collected from users and which were used in the benchmark.

This could also give more insight into the general problems of main memory-based graph search and not only the ones specific for graphANNIS.

A clear result of the comparison with relANNIS in Section 7.1.2 was that there is a problematic class of queries which has a large influence on the overall performance. These are queries that contain both high and low selectivity annotation searches, but where the AQL operator is defined to use the low-selectivity query as LHS. This can result in query plans where the source iterator in the plan yields a lot of results that need to be check if they fulfill the AQL operator condition. A solution to this problem is to have inverse operators for each non-commutative AQL operator and store inverse edges in the graph storages. The result of the benchmark suggests that the speedup in relation to relANNIS could be almost doubled if this feature would be supported.

8.4.2. Parallelization

One of the areas in the evaluation with mixed results was the impact of parallelization. While thread-based parallelization of joins had a notable effect, the SIMD-based approach was less helpful. It would be interesting to compare the approaches for parallelization of more recent versions of PostgreSQL with the ones of graphANNIS, to find more possibilities for thread-based parallelization in graphANNIS and to directly compare their efficiency in a benchmark. Also, the current benchmarks used only very few cores. Since modern server systems have more CPUs available, it should be tested which parallelization techniques are able to scale on the number of CPUs. Also, the behavior of the CPUs should be measured in more detail and more systematic. While there have been ad-hoc measurements in the development process to find performance-critical parts of the application, more systematic measurements of, for example, cache misses could provide more insight into the effects of parallelization and how to better optimize such a query system, like it was done for the cache-sensitive skip list implementation in Sprenger et al. (2017).

Other approaches to parallelization could be to exploit the partitioning of the data into graph components. Currently, only different annotation layers are partitioned into separate graph storages. If a graph inside a graph storage consists of multiple strongly connected components (Cormen et al. 2009, p. 1171), these could be stored separately and queried in parallel. For example, if a graph storage represents a syntax tree, each sentence will be a strongly connected component. Treebank query systems like trgrep2 (Rohde 2005) or TIGERSearch (Lezius 2002) are already able to use this partitioning of sentences. If implemented in graphANNIS based on strongly connected components, this optimization could be applied to all annotation types with a similar structure.

8.4.3. Query language support

The design of graphANNIS allows adding new operator implementations, which could be used to add new features to AQL but also to provide alternative implementations for existing AQL operators. These could be specialized in certain types of corpora, similar to the specialized graph storages. Another possibility would be the support for different query languages. For example, CQP is a popular query language that

users might already know (Evert and Hardie 2011). Also, there is a movement to design a generic query language as an ISO standard (Banski et al. 2016). Support for more queries could be achieved by mapping the queries to the same internal data structures that are described in Section 5.2. Since KorAP also supports multiple query languages (Diewald and Margaretha 2016), including a subset of AQL, this would make it easier to compare both systems in a benchmark using the same set of queries. Given the modularity of both the ANNIS and KorAP architectures, support for additional query languages could also make it easier to integrate graphANNIS as a back-end for different existing user interfaces and to combine it with other query execution systems behind the same user interface of a web-service.

8.4.4. Support for more domains

Another possible extension of graphANNIS is to explore how its design can be used to create a graph query system for other domains, with graphs of similar size and where users typically query a read-only graph, too. This would need more flexible support for components and a different set of domain-specific operators. GraphANNIS is much more suited than the original relANNIS implementation for such an extension because its data model is more generic and extensible. Also, partitioning by edge components instead of documents allows much more flexible document structures, and this can be useful for other domains as well.

One domain where such flexible structures are needed is the study of text reuse phenomena. Text reuse can have multiple forms, "that range from quotations to allusions and translation" (Berti et al. 2014, p. 1). To study such reuse, text fragments need to be linked with other texts, text fragments or external information like named entity or geographical databases. The fragments itself often do not belong to just one document but to several ones, and the documents also have more complex relationships than in the traditional corpus/sub-corpus model. In Berti et al. (2014) RDF is used to model these connections. Representing these links as part of graphANNIS would allow adding additional kind of annotations and to perform analysis based on the combination of these annotations. The design of graphANNIS should allow querying this kind of corpora as fast as more conservatively structured ones. An open issue in such a scenario of highly linked texts is if all data needs to be located on the same server. The current single server system might be well suited to store an even larger number of texts, but for example, copying all the linked information from external databases as explicit annotations might be impossible due to the recursive nature of the links or legal limitations. A federated search infrastructure like the one proposed as part of the "Common Language Resources and Technology Infrastructure" (CLARIN) organization (Stehouwer et al. 2012) could be used to identify relevant documents, and to fetch more complete data-sets on-demand for a specific query.

8.5. Summary

This dissertation presented graphANNIS, a design and implementation of a linguistic corpus query system based on graphs. It supports the existing and popular query

language AQL and is based on the generic Salt data model, but its implementation features novel approaches to allow efficient query execution. One of these design decisions was to partition the corpora into edge components and to provide specialized implementations called graph storages. These graph storages efficiently implement finding reachable nodes and are optimized for different types of graphs. The query system decides which graph storage implementation is used for a specific annotation graph based on its properties at runtime, which allows the system to adapt to different kinds of annotation graphs. Representing such multi-layer corpora has already been possible with the predecessor query system relANNIS, which is based on the relational PostgreSQL database. GraphANNIS does not only allow to represent these multi-layer corpora but actually exploits the different annotations of the different layers to provide faster query execution. Evaluation has shown, that graphANNIS is about ten times faster for a realistic workload than the original relANNIS implementation. The about 4000 queries of the workload have been made publicly available to make it easier to compare linguistic query system implementations. Despite the performance improvements, graphANNIS has a much more flexible data model than its predecessor. Adding new types of annotations, extending the query language and using novel ways of connecting annotations from different corpora is much easier due to this flexibility. GraphANNIS represents a good example use-case for how to implement efficient graph queries for a specific domain, and the results of this work have very practical relevance for corpus linguistic research.

A. Appendix

A.1. Data-sets and software

All relevant software and data-sets used in the thesis have been uploaded to the Zenodo[1] repository. This includes the following artifacts:

- source code of graphANNIS: `http://doi.org/10.5281/zenodo.1146565`
- source code of relANNIS: `https://doi.org/10.5281/zenodo.1161400`
- AQL queries, benchmark results and scripts used in the evaluation: `http://doi.org/10.5281/zenodo.1161374`
- corpora in the relANNIS data format that have been used in the benchmarks: `http://doi.org/10.5281/zenodo.1161383` (different open-access corpora) and `http://doi.org/10.5281/zenodo.1161390` (corpora restricted to academic use)

A.2. Known ANNIS servers

Table A.1 contains a list of ANNIS servers the author is aware of. Since ANNIS can be used also as desktop-version or projects can decide to not publicly announce their servers, this list is incomplete. Also, not all of these instances provide access to their corpora without login, which makes it difficult to estimate how many corpora a hosted on a specific server. See for example the curated list of corpora that are hosted at the Georgetown University [2]. Access to many of these corpora is restricted and thus they will not be visible in the ANNIS interface without login. The list in Table A.1 also contains outdated versions of ANNIS, which have been included because even if projects do not update their versions (for example after the projects ends), they still contain relevant data that is accessible in ANNIS.

[1] `https://zenodo.org/` (last accessed 2018-01-27)
[2] `https://corpling.uis.georgetown.edu/annis-corpora/` (last accessed 2018-01-27)

Institution	URL
Eurac Research	https://commul.eurac.edu/annis/didi/
Georgetown University	https://corpling.uis.georgetown.edu/annis
Humboldt-Universität zu Berlin	https://korpling.org/annis3/
Hamburger Zentrum für Sprachkorpora	http://annis.corpora.uni-hamburg.de/gui/
IDS Mannheim	http://clarin.ids-mannheim.de/SFB632/A6/Annis-web/
Indiana University	http://nlp.indiana.edu:8085/annis-gui-3.1.7/
MERLIN project	http://merlin-platform.eu/annis/
Perseus Digital Library	http://annis.perseus.tufts.edu/Annis-web/
Regensburg Russian Diachronic Corpus	http://rhssl1.uni-regensburg.de:8888/Annis-web
Ruhr-Universität Bochum	https://www.linguistics.rub.de/annis/annis3/REM/
SMS4Science	http://server.linguistik.uzh.ch:8080/annis-gui/
The Language Archive	https://corpus1.mpi.nl/ds/annis/
Universität Frankfurt	http://corpora.acoli.informatik.uni-frankfurt.de:8080/annis/
Universität Tübingen	http://sifnos.sfs.uni-tuebingen.de/annis/

Table A.1.: Incomplete list of public visible ANNIS-servers. All URLs have been last accessed at 2018-01-24.

Bibliography

Ágel, Vilmos and Mathilde Hennig (May 2014). *Kasseler Junktionskorpus (Version 1.1)*. Justus-Liebig-Universität Gießen. http://hdl.handle.net/11022/0000-0000-2102-8 (cit. on p. 99).

Aho, Alfred V. and Jeffrey D. Ullman (1979). "Universality of Data Retrieval Languages". In: *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '79. San Antonio, Texas: ACM, pp. 110–119. DOI: 10.1145/567752.567763 (cit. on p. 11).

Arenas, Marcelo, Sebastián Conca, and Jorge Pérez (2012). "Counting beyond a Yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard". In: *Proceedings of the 21st international conference on World Wide Web*. ACM, pp. 629–638 (cit. on p. 12).

Bański, Piotr, Joachim Bingel, Nils Diewald, Elena Frick, Michael Hanl, Marc Kupietz, Piotr Pzik, Carsten Schnober, and Andreas Witt (Dec. 2013). "KorAP: the new corpus analysis platform at IDS Mannheim". In: *Human Language Technology Challenges for Computer Science and Linguistics : 6th language & technology conference*. Ed. by Zygmunt Vetulani and Hans Uszkoreit. Poznań, Poland, pp. 586–587. URL: http://ids-pub.bsz-bw.de/frontdoor/index/index/docId/3261 (cit. on pp. 13, 17).

Banski, Piotr, Elena Frick, and Andreas Witt (May 2016). "Corpus Query Lingua Franca (CQLF)". In: *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC 2016)*. Ed. by Nicoletta Calzolari (Conference Chair), Khalid Choukri, Thierry Declerck, Sara Goggi, Marko Grobelnik, Bente Maegaard, Joseph Mariani, Helene Mazo, Asuncion Moreno, Jan Odijk, and Stelios Piperidis. Portorož, Slovenia: European Language Resources Association (ELRA). ISBN: 978-2-9517408-9-1. URL: http://www.lrec-conf.org/proceedings/lrec2016/pdf/1137_Paper.pdf (cit. on p. 124).

Baroni, Marco (2009). "Corpus Linguistics". In: ed. by Anke Lüdeling and Merja Kytö. Vol. 2. Handbooks of Linguistics and Communication Science (HSK) 29. Mouton de Gruyter. Chap. Distributions in text, pp. 803–822 (cit. on p. 52).

Berti, Monica, Bridget Almas, David Dubin, Greta Franzini, Simona Stoyanova, and Gregory R Crane (2014). "The Linked Fragment: TEI and the encoding of text reuses of lost authors". In: *Journal of the Text Encoding Initiative* 8. DOI: 10.4000/jtei.1218 (cit. on p. 124).

Białecki, Andrzej, Robert Muir, Grant Ingersoll, and Lucid Imagination (Aug. 2012). "Apache Lucene 4". In: *SIGIR 2012 workshop on open source information retrieval*. Portland, USA. URL: http://opensearchlab.otago.ac.nz/paper_10.pdf (cit. on p. 17).

Biber, Douglas, Susan Conrad, and Randi Reppen (1998). *Corpus Linguistics: Investigating Language Structure and Use*. Ed. by Jean Aitchison and Rupert Murdoch. Cambridge Approaches to Linguistics. Cambridge University Press. DOI: 10.1017/CBO9780511804489 (cit. on p. 1).

Bies, Ann, Mark Ferguson, Karen Katz, Robert MacIntyre, Victoria Tredinnick, Grace Kim, Mary Ann Marcinkiewicz, and Britta Schasberger (1995). *Bracketing guidelines for Treebank II style Penn Treebank project*. Tech. rep. University of Pennsylvania. URL: http://languagelog.ldc.upenn.edu/myl/PennTreebank1995.pdf (cit. on p. 9).

Bird, Steven and Mark Liberman (2001). "A formal framework for linguistic annotation". In: *Speech Communication* 33.1–2. Speech Annotation and Corpus Tools, pp. 23–60. ISSN: 0167-6393. DOI: 10.1016/S0167-6393(00)00068-6 (cit. on p. 10).

Bodmer, Franck (1996). "Abfragekomponente von COSMAS-II". In: *LDV-Info 1996*. 8. Institut für Deutsche Sprache. ISBN: 3-922641-41-5. URL: https://ids-pub.bsz-bw.de/frontdoor/index/index/docId/5884 (cit. on p. 17).

Brants, Sabine, Stefanie Dipper, Peter Eisenberg, Silvia Hansen-Schirra, Esther König, Wolfgang Lezius, Christian Rohrer, George Smith, and Hans Uszkoreit (2004). "TIGER: Linguistic Interpretation of a German Corpus". In: *Research on Language and Computation* 2.4, pp. 597–620. ISSN: 1572-8706. DOI: 10.1007/s11168-004-7431-3 (cit. on pp. 4–6, 15, 99).

Brouwer, Matthijs, Hennie Brugman, and Marc Kemps-Snijders (2017). "MTAS: A Solr/Lucene based Multi Tier Annotation Search solution". In: *Selected papers from the CLARIN Annual Conference 2016*. Ed. by Lars Borin. Linköping Electronic Conference Proceedings 136. Linköping University Electronic Press, Linköpings universitet, pp. 19–37. URL: http://www.ep.liu.se/ecp/136/002/ecp17136002.pdf (cit. on p. 17).

Carroll, Jeremy, Ian Dickinson, Chris Dollin, Dave Reynolds, Andy Seaborne, and Kevin Wilkinson (2004). "Jena: Implementing the Semantic Web Recommendations". In: *Proceedings of the 13th International World Wide Web Conference on Alternate Track Papers & Posters*. WWW Alt. '04. New York, NY, USA: ACM, pp. 74–83. ISBN: 1-58113-912-8. DOI: 10.1145/1013367.1013381 (cit. on p. 12).

Carroll, Jeremy and Graham Klyne (Feb. 2004). *Resource Description Framework (RDF): Concepts and Abstract Syntax*. W3C Recommendation. W3C. URL: http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/ (cit. on p. 11).

Chiarcos, Christian (2012). "POWLA: Modeling Linguistic Corpora in OWL/DL". In: *The Semantic Web: Research and Applications: 9th Extended Semantic Web Conference, ESWC 2012, Heraklion, Crete, Greece, May 27-31, 2012. Proceedings*. Ed. by Elena Simperl, Philipp Cimiano, Axel Polleres, Oscar Corcho, and Valentina Presutti. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 225–239. ISBN: 978-3-642-30284-8. DOI: 10.1007/978-3-642-30284-8_22 (cit. on p. 12).

Chiarcos, Christian, John McCrae, Philipp Cimiano, and Christiane Fellbaum (2013). "Towards Open Data for Linguistics: Linguistic Linked Data". In: *New Trends of Research in Ontologies and Lexical Resources: Ideas, Projects, Systems*. Ed. by Alessandro Oltramari, Piek Vossen, Lu Qin, and Eduard Hovy. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 7–25. ISBN: 978-3-642-31782-8. DOI: 10.1007/978-3-642-31782-8_2 (cit. on p. 12).

Christ, Oliver (1994). “A Modular and Flexible Architecture for an Integrated Corpus Query System”. In: *Proceedings of COMPLEX ’94*. Budapest, Hungary, pp. 23–32. URL: https://arxiv.org/pdf/cmp-lg/9408005.pdf (cit. on p. 15).

Codd, E. F. (1972). “Relational completeness of data base sublanguages”. In: *Database Systems*. Prentice-Hall, pp. 65–98 (cit. on p. 43).

Coniglio, Marco, Karin Donhauser, and Eva Schlachter (2014). *HIPKON: Historisches Predigtenkorpus zum Nachfeld (Version 1.0)*. Humboldt-Universität zu Berlin. SFB 632 Teilprojekt B4. URL: http://hdl.handle.net/11022/0000-0000-2D18-4 (cit. on p. 99).

Consens, Mariano P and Alberto O Mendelzon (1990). “GraphLog: a visual formalism for real life recursion”. In: *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. ACM, pp. 404–416 (cit. on p. 11).

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein (2009). *Introduction to Algorithms*. 3rd ed. The MIT Press. ISBN: 9780262533058 (cit. on pp. 15, 37, 48, 58, 60, 64, 70, 93, 123).

Deursen, Arie van, Paul Klint, and Joost Visser (June 2000). “Domain-specific Languages: An Annotated Bibliography”. In: *SIGPLAN Not.* 35.6, pp. 26–36. ISSN: 0362-1340. DOI: 10.1145/352029.352035 (cit. on p. 14).

Diewald, Nils and Eliza Margaretha (2016). “Krill: KorAP search and analysis engine”. In: *JLCL* 31.1, pp. 73–90. URL: http://www.jlcl.org/2016_Heft1/jlcl-2016-1-4DiewaldMargaretha.pdf (cit. on pp. 17, 121, 124).

Dipper, Stefanie (2005). “XML-based Stand-off Representation and Exploitation of Multi-Level Linguistic Annotation.” In: *Berliner XML Tage*, pp. 39–50. URL: http://pub.sfb632.uni-potsdam.de/publications/D1/D1_Dipper_2005a.pdf (cit. on pp. 7, 9, 16).

– (2008). “Corpus Linguistics”. In: ed. by Anke Lüdeling and Merja Kytö. Vol. 1. Handbooks of Linguistics and Communication Science (HSK) 29. Mouton de Gruyter. Chap. Theory-driven and corpus-driven computational linguistics, and the use of corpora, pp. 68–96 (cit. on p. 2).

Donhauser, Karin, Jost Gippert, and Rosemarie Lühr (May 2015). *Deutsch Diachron Digital - Referenzkorpus Altdeutsch (Version 0.1)*. Humboldt-Universität zu Berlin. URL: http://hdl.handle.net/11022/0000-0000-7FC2-7 (cit. on p. 99).

Dries, Anton, Siegfried Nijssen, and Luc De Raedt (2009). “A query language for analyzing networks”. In: *Proceedings of the 18th ACM conference on Information and knowledge management*. ACM, pp. 485–494 (cit. on p. 11).

Druskat, Stephan, Lennart Bierkandt, Volker Gast, Christoph Rzymski, and Florian Zipser (2014). “Atomic: An open-source software platform for multi-level corpus annotation”. In: *Proceedings of the 12th Konferenz zur Verarbeitung natürlicher Sprache (KONVENS 2014)*. Vol. 1. Hildesheim, pp. 228–234. URL: http://nbn-resolving.de/urn:nbn:de:gbv:hil2-opus-2866 (cit. on p. 120).

Druskat, Stephan, Volker Gast, Thomas Krause, and Florian Zipser (May 2016). “corpus-tools.org: An Interoperable Generic Software Tool Set for Multi-layer Linguistic Corpora”. In: *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC 2016)*. Ed. by Nicoletta Calzolari (Conference Chair), Khalid Choukri, Thierry Declerck, Sara Goggi, Marko Grobelnik, Bente Maegaard, Joseph Mariani, Helene Mazo, Asuncion Moreno, Jan Odijk, and Stelios Piperidis.

Portorož, Slovenia: European Language Resources Association (ELRA). ISBN: 978-2-9517408-9-1. URL: http://www.lrec-conf.org/proceedings/lrec2016/pdf/918_Paper.pdf (cit. on p. 119).

Druskat, Stephan, Thomas Krause, Carolin Odebrecht, and Florian Zipser (Mar. 2017). "Agile creation of multi-layer corpora with corpus-tools.org". In: *DGfS-CL Poster Session. 39. Jahrestagung der Deutschen Gesellschaft für Sprachwissenschaft (DGfS)*. Saarbrücken. URL: http://dgfs2017.uni-saarland.de/wordpress/abstracts/clposter/cl_17_drus.pdf (cit. on p. 120).

Erling, Orri (2012). "Virtuoso, a Hybrid RDBMS/Graph Column Store." In: *IEEE Data Engineering Bulletin* 35.1, pp. 3–8 (cit. on p. 14).

Evert, Stefan and Andrew Hardie (2011). "Twenty-first century Corpus Workbench: Updating a query architecture for the new millennium". In: *Proceedings of the Corpus Linguistics 2011 conference*. University of Birmingham. URL: http://eprints.lancs.ac.uk/62721/ (cit. on pp. 14, 15, 120, 124).

– (2015a). *Ziggurat data model and file format, version 1.0*. URL: http://cwb.sourceforge.net/files/Ziggurat%20data%20model%201.0.pdf (cit. on p. 18).

– (2015b). "Ziggurat: A new data model and indexing format for large annotated text corpora". In: *Challenges in the Management of Large Corpora (CMLC-3)*, p. 21. URL: http://ids-pub.bsz-bw.de/files/3826/Evert_Hardie_Ziggurat_A_new_data_model_and_indexing_format_2015.pdf (cit. on p. 18).

Fielding, Roy Thomas (2000). "Architectural Styles and the Design of Network-based Software Architectures". PhD thesis. University of California, Irvine. URL: http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf (cit. on p. 45).

Frick, Elena, Carsten Schnober, and Piotr Banski (2012). "Evaluating Query Languages for a Corpus Processing System." In: *LREC*, pp. 2286–2294. URL: http://www.lrec-conf.org/proceedings/lrec2012/pdf/800_Paper.pdf (cit. on p. 14).

Garcia-Molina, Hector, Jeffrey D. Ullman, and Jennifer Widom (2000). *Database System Implementation*. Prentice Hall. ISBN: 0-13-040264-8 (cit. on pp. 47, 48, 82, 90).

Goschler, Juliana (2014). "Variation im Kernbereich: Koordinierte Subjekte und Subjekt-Verb-Kongruenz im Deutschen". In: *Zwischen Kern und Peripherie: Untersuchungen zu Randbereichen in Sprache und Grammatik*. Ed. by Antonio Machicao y Priemer, Andreas Nolda, and Athina Sioupi. Studia grammatica 76. Berlin: De Gruyter, pp. 89–101 (cit. on pp. 4, 7).

Grün, Christian (2006). "Pushing XML Main Memory Databases to their Limits". In: *Tagungsband zum 18. GI-Workshop über Grundlagen von Datenbanken (18th GI-Workshop on the Foundations of Databases), Wittenberg, Sachsen-Anhalt, 6.-9. Juni 2006*, pp. 60–64. URL: http://dbs.informatik.uni-halle.de/GvD2006/gvd06_gruen.pdf (cit. on p. 16).

Grust, Torsten, Maurice Van Keulen, and Jens Teubner (2004). "Accelerating XPath evaluation in any RDBMS". In: *ACM Transactions on Database Systems (TODS)* 29.1, pp. 91–131. URL: http://dl.acm.org/citation.cfm?id=974754 (cit. on pp. 14, 16, 32).

Gubichev, Andrey, Technische Universität München, Thomas Neumann, and Technische Universität München (2011). "Path Query Processing on Very Large RDF Graphs". In: *WebDB*. Athens, Greece (cit. on p. 12).

Hanke, Thomas, Jakob Storz, and Sven Wagner (May 2010). "iLex: Handling multicamera recordings". In: *LREC 2010. 7th International Conference on Language Resources and Evaluation. Workshop Proceedings. W13. 4th Workshop on Representation and Processing of Sign Languages: Corpora and Sign Language Technologies*, pp. 110–111. URL: http://www.sign-lang.uni-hamburg.de/dgs-korpus/files/inhalt_pdf/Hanke_et_al_2010_MultiCameras.pdf (cit. on p. 2).

Hardie, Andrew (2012). "CQPweb — combining power, flexibility and usability in a corpus analysis tool". In: *International Journal of Corpus Linguistics* 17.3, pp. 380–409. DOI: 10.1075/ijcl.17.3.04har (cit. on p. 15).

Harris, Steven and Andy Seaborne (Mar. 2013). *SPARQL 1.1 Query Language*. W3C Recommendation. W3C. URL: http://www.w3.org/TR/2013/REC-sparql11-query-20130321/ (cit. on p. 12).

Hirschmann, Hagen, Anke Lüdeling, and Amir Zeldes (2008). "What's hard? - Quantitative evidence for difficult constructions in German learner data". In: *Proceedings of QITL 3. Helsinki*. http://edoc.hu-berlin.de/docviews/abstract.php?lang=ger&id=37129 (cit. on p. 99).

Hütter, Karsten (2008). "Entwicklung einer Benutzerschnittstelle für die Suche in linguistischen Mehrebenen-Korpora unter Betrachtung softwareergonomischer Gesichtspunkte". MA thesis. Humboldt-Universität zu Berlin (cit. on p. 16).

Ide, Nancy and Laurent Romary (2004). "International standard for a linguistic annotation framework". In: *Natural language engineering* 10.3-4, pp. 211–225 (cit. on p. 10).

Ide, Nancy and Keith Suderman (2007). "GrAF: A Graph-based Format for Linguistic Annotations". In: *Proceedings of the Linguistic Annotation Workshop*. LAW '07. Prague, Czech Republic: Association for Computational Linguistics, pp. 1–8. URL: http://dl.acm.org/citation.cfm?id=1642059.1642060 (cit. on p. 10).

ISO:24612 (June 2012). *Language resource management—Linguistic annotation framework (LAF)*. URL: https://www.iso.org/standard/37326.html (cit. on p. 10).

Krause, Thomas, Ulf Leser, and Anke Lüdeling (2016). "graphANNIS: A Fast Query Engine for Deeply Annotated Linguistic Corpora". In: *JLCL* 31.1, pp. iii–25. URL: http://www.jlcl.org/2016_Heft1/jlcl-2016-1-1KrauseEtAl.pdf (cit. on p. 118).

Krause, Thomas, Anke Lüdeling, Carolin Odebrecht, Laurent Romary, Peter Schirmbacher, and Dennis Zielke (2014). "LAUDATIO-Repository: Accessing a heterogeneous field of linguistic corpora with the help of an open access repository". In: *Digital Humanities 2014 Conference. Poster Session*. URL: http://dharchive.org/paper/DH2014/Poster-34.xml (cit. on p. 117).

Krause, Thomas, Anke Lüdeling, Carolin Odebrecht, and Amir Zeldes (2012). "Multiple Tokenization in a Diachronic Corpus". In: *Exploring Ancient Languages through Corpora Conference (EALC)*. Universitetet i Oslo. URL: http://www.hf.uio.no/ifikk/english/research/projects/proiel/ealc/abstracts/Krause_et_al.pdf (cit. on p. 24).

Krause, Thomas and Amir Zeldes (2016). "ANNIS3: A new architecture for generic corpus query and visualization". In: *Digital Scholarship in the Humanities* 31.1, pp. 118–139. ISSN: 2055-7671. DOI: 10.1093/llc/fqu057 (cit. on pp. 1, 8, 16, 26).

Kretz, Matthias (2015). "Extending C++ for explicit data-parallel programming via SIMD vector types". PhD thesis. Frankfurt am Main: Goethe-Universität Frankfurt am Main, p. 256. URL: http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:hebis:30:3-384155 (cit. on p. 94).

Kübler, Sandra and Heike Zinsmeister (Dec. 18, 2014). *Corpus linguistics and linguistically annotated corpora*. Bloomsbury Publishing. ISBN: 9781441116758 (cit. on p. 117).

Kupietz, Marc and Harald Lüngen (May 2014). "Recent Developments in DeReKo". In: *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14)*. Ed. by Nicoletta Calzolari (Conference Chair), Khalid Choukri, Thierry Declerck, Hrafn Loftsson, Bente Maegaard, Joseph Mariani, Asuncion Moreno, Jan Odijk, and Stelios Piperidis. Reykjavik, Iceland: European Language Resources Association (ELRA). ISBN: 978-2-9517408-8-4. URL: http://www.lrec-conf.org/proceedings/lrec2014/pdf/842_Paper.pdfs (cit. on pp. 17, 121).

Leser, Ulf (2005). "A query language for biological networks". In: *Bioinformatics* 21.Suppl. 2, pp. ii33–ii39. DOI: 10.1093/bioinformatics/bti1105 (cit. on p. 11).

Lezius, Wolfgang (2002). "TIGERSearch Ein Suchwerkzeug für Baumbanken". In: *Tagungsband zur Konvens* (cit. on pp. 15, 26, 123).

Losemann, Katja and Wim Martens (2012). "The complexity of evaluating path expressions in SPARQL". In: *Proceedings of the 31st symposium on Principles of Database Systems*. ACM, pp. 101–112 (cit. on p. 12).

Lüdeling, Anke, Maik Walter, Emil Kroymann, and Peter Adolphs (2005). "Multi-level error annotation in learner corpora". In: *In Proceedings of Corpus Linguistics 2005* (cit. on p. 7).

Lühr, Rosemarie, Vera Faßhauer, Daniela Prutscher, and Henry Seidel (June 2015). *Fuerstinnenkorrespondenz 1.1*. Universität Jena, DFG. http://hdl.handle.net/11022/0000-0000-82A0-7 (cit. on p. 99).

Marcus, Mitchell P., Mary Ann Marcinkiewicz, and Beatrice Santorini (June 1993). "Building a Large Annotated Corpus of English: The Penn Treebank". In: *Comput. Linguist.* 19.2, pp. 313–330. ISSN: 0891-2017. URL: http://dl.acm.org/citation.cfm?id=972470.972475 (cit. on p. 15).

Martens, Scott (2012). "TüNDRA: TIGERSearch-style treebank querying as an XQuery-based web service". In: *Proceedings of the joint CLARIN-D/DARIAH Workshop "Service-oriented Architectures (SOAs) for the Humanities: Solutions and Impacts"*. Digital Humanities Conference. URL: http://www.clarin-d.de/images/workshops/proceedingssoasforthehumanities.pdf (cit. on p. 16).

Marton, József, Gábor Szárnyas, and Dániel Varró (2017). "Formalising openCypher Graph Queries in Relational Algebra". In: *Advances in Databases and Information Systems: 21st European Conference, ADBIS 2017, Nicosia, Cyprus, September 24-27, 2017, Proceedings*. Ed. by Mārīte Kirikova, Kjetil Nørvåg, and George A. Papadopoulos. Cham: Springer International Publishing, pp. 182–196. ISBN: 978-3-319-66917-5. DOI: 10.1007/978-3-319-66917-5_13 (cit. on pp. 13, 42).

Mendelzon, Alberto O and Peter T Wood (1995). “Finding regular simple paths in graph databases”. In: *SIAM Journal on Computing* 24.6, pp. 1235–1258 (cit. on p. 11).

Mengel, Andreas and Wolfgang Lezius (2000). “An XML-based Representation Format for Syntactically Annotated Corpora.” In: *Proceedings of the Second International Conference on Language Resources and Engineering (LREC 2000)*. Athens, Greece. URL: http://www.amengel.de/pubs/am-wl_lrec2000.pdf (cit. on p. 9).

Mernik, Marjan, Jan Heering, and Anthony M. Sloane (Dec. 2005). “When and How to Develop Domain-specific Languages”. In: *ACM Comput. Surv.* 37.4, pp. 316–344. ISSN: 0360-0300. DOI: 10.1145/1118890.1118892 (cit. on p. 27).

Meurer, Paul (2012). “Exploring newspaper language: using the web to create and investigate a large corpus of modern Norwegian”. In: ed. by Gisle Andersen. Benjamins. Chap. Corpuscle – a new corpus management platform for annotated corpora, pp. 31–49 (cit. on p. 122).

Meyer, Charles F. (2002). *English corpus linguistics: An introduction*. Cambridge University Press (cit. on p. 1).

– (2008). “Corpus Linguistics”. In: ed. by Anke Lüdeling and Merja Kytö. Vol. 1. Handbooks of Linguistics and Communication Science (HSK) 29. Mouton de Gruyter. Chap. Pre-electronic corpora, pp. 1–14 (cit. on p. 1).

Nambiar, Raghunath Othayoth, Matthew Lanken, Nicholas Wakou, Forrest Carman, and Michael Majdalany (2009). “Transaction Processing Performance Council (TPC): Twenty Years Later – A Look Back, a Look Ahead”. In: *Performance Evaluation and Benchmarking*. Ed. by Raghunath Nambiar and Meikel Poess. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 1–10. ISBN: 978-3-642-10424-4 (cit. on p. 98).

Neumann, Thomas and Gerhard Weikum (2010). “The RDF-3X engine for scalable management of RDF data”. In: *The VLDB Journal* 19.1, pp. 91–113 (cit. on p. 12).

Nivre, Joakim (2008). “Corpus Linguistics”. In: ed. by Anke Lüdeling and Merja Kytö. Vol. 1. Handbooks of Linguistics and Communication Science (HSK) 29. Mouton de Gruyter. Chap. Treebanks, pp. 225–241 (cit. on p. 15).

Odebrecht, Carolin (2012). “Lexical Bundles. Eine korpuslinguistische Untersuchung”. MA thesis. Humboldt-Universität zu Berlin, Philosophische Fakultät II. DOI: 10.18452/14164 (cit. on p. 99).

– (2017). “MKM – ein Metamodell für Korpusmetadaten”. PhD thesis. Humboldt-Universität zu Berlin (cit. on p. 23).

Odebrecht, Carolin, Malte Belz, Amir Zeldes, Anke Lüdeling, and Thomas Krause (Sept. 2017). “RIDGES Herbology: designing a diachronic multi-layer corpus”. In: *Language Resources and Evaluation* 51.3, pp. 695–725. DOI: 10.1007/s10579-016-9374-3 (cit. on pp. 7, 24, 99).

Ordonez, Carlos (2010). “Optimization of linear recursive queries in SQL”. In: *Knowledge and Data Engineering, IEEE Transactions on* 22.2, pp. 264–277 (cit. on p. 14).

Pęzik, Piotr (2011). “Providing Corpus Feedback for Translators with the PELCRA Search Engine for NKJP”. In: *Explorations across Languages and Corpora: PALC 2009*. Ed. by Stanislaw Gozdz-Roszkowski. Łódź Studies in Linguistics. Frankfurt am Main; New York: Peter Lang, pp. 135–144. URL: http://nkjp.pl/settings/papers/32Pezik-ok.pdf (cit. on p. 17).

Pęzik, Piotr (Oct. 2013). "Indexed graph databases for querying rich TEI annotation". In: *Perspectives on querying TEI-annotated data (Workshop at the TEI Conference and Members Meeting)*. Ed. by Piotr Banski, Marc Kupietz, and Andreas Witt. Rome, Italy. URL: http://www.tei-c.org/Vault/MembersMeetings/2013/wp-content/uploads/2013/09/Pezik.pdf (cit. on p. 13).

Piatetsky-Shapiro, Gregory and Charles Connell (1984). "Accurate estimation of the number of tuples satisfying a condition". In: *ACM Sigmod Record* 14.2, pp. 256–276 (cit. on p. 55).

Poosala, Viswanath, Peter J Haas, Yannis E Ioannidis, and Eugene J Shekita (1996). "Improved histograms for selectivity estimation of range predicates". In: *ACM Sigmod Record*. Vol. 25. 2. ACM, pp. 294–305 (cit. on p. 56).

Przepiórkowski, Adam, Zygmunt Krynicki, Lukasz Debowski, Marcin Wolinski, Daniel Janus, and Piotr Banski (May 2004). "A Search Tool for Corpora with Positional Tagsets and Ambiguities." In: *Proceedings of the Fourth International Conference on Language Resources and Evaluation (LREC 2004)*. Lisbon, Portugal. URL: http://www.lrec-conf.org/proceedings/lrec2004/pdf/275.pdf (cit. on p. 17).

Rábara, Radoslav, Pavel Rychlý, Ondrej Herman, and Miloš Jakubícek (July 2017). "Accelerating corpus search using multiple cores". In: *Proceedings of the Workshop on Challenges in the Management of Large Corpora and Big Data and Natural Language Processing (CMLC-5+BigNLP)*. Birmingham, p. 30. URL: https://ids-pub.bsz-bw.de/files/6243/7.+Rabara_etal_Accelerating_Corpus_Search_2017.pdf (cit. on p. 122).

Reynaert, Martin, Matje van de Camp, and Menno van Zaanen (Aug. 2014). "OpenSoNaR: user-driven development of the SoNaR corpus interfaces". In: *The 25th International Conference on Computational Linguistics (System Demonstrations)*. Dublin, Ireland, pp. 124–128. URL: http://www.aclweb.org/anthology/C/C14/C14-2027.pdf (cit. on p. 17).

Reznicek, Marc, Anke Lüdeling, Cedric Krummes, Franziska Schwantuschke, Maik Walter, Karin Schmidt, Hagen Hirschmann, and Torsten Andreas (2012). *Das Falko-Handbuch. Korpusaufbau und Annotationen Version 2.01*. Tech. rep. https://www.linguistik.hu-berlin.de/de/institut/professuren/korpuslinguistik/forschung/falko/FalkoHandbuchV2/at_download/file. Technical report, Department of German Studies and Linguistics, Humboldt University, Berlin, Germany (cit. on p. 99).

Robinson, Ian, J Webber, and E Eifrem (2013). *Graph Databases*. O'Reilly Media (cit. on pp. 13, 26, 119).

Rodriguez, Marko A. (2015). "The Gremlin Graph Traversal Machine and Language (Invited Talk)". In: *Proceedings of the 15th Symposium on Database Programming Languages*. DBPL 2015. Pittsburgh, PA, USA: ACM, pp. 1–10. ISBN: 978-1-4503-3902-5. DOI: 10.1145/2815072.2815073 (cit. on p. 13).

Rodriguez, Marko A. and Peter Neubauer (2010). "The Graph Traversal Pattern". In: *CoRR* abs/1004.1001. URL: http://arxiv.org/abs/1004.1001 (cit. on pp. 13, 42).

Rodríguez, Kepa Joseba, Stefanie Dipper, Michael Götze, Massimo Poesio, Giuseppe Riccardi, Christian Raymond, and Joanna Wisniewska (2007). "Standoff Coordination for Multi-tool Annotation in a Dialogue Corpus". In: *Proceedings of the*

Linguistic Annotation Workshop. LAW '07. Prague, Czech Republic: Association for Computational Linguistics, pp. 148–155. URL: http://dl.acm.org/citation.cfm?id=1642059.1642083 (cit. on p. 7).

Rohde, Douglas LT (2005). *Tgrep2 user manual*. URL: http://tedlab.mit.edu/~dr/Tgrep2/tgrep2.pdf (cit. on pp. 15, 123).

Rosenfeld, Viktor (2010). *An implementation of the Annis 2 query language*. Tech. rep. Humboldt-Universität zu Berlin. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.403.1104 (cit. on pp. 8, 16, 26, 32, 62, 122).

– (2012). "A Linguistic Query Language On Top Of A Column-Oriented Main-Memory Database". MA thesis. Humboldt-Universität zu Berlin. URL: http://www.user.tu-berlin.de/viktor-rosenfeld/assets/publications/diplomarbeit.pdf (cit. on pp. 16, 26, 33, 118, 119, 122).

Sauer, Simon (2013). *BeMaTaC*. URL: http://u.hu-berlin.de/bematac (cit. on p. 99).

Schäfer, Roland (2015). "Processing and querying large web corpora with the COW14 architecture". In: *Challenges in the Management of Large Corpora (CMLC-3)*, pp. 28–34. URL: http://ids-pub.bsz-bw.de/files/3826/Schaefer_Processing_and_querying_large_web_corpora_2015.pdf (cit. on p. 117).

Schiller, Anne, Simone Teufel, Christine Stöckert, and Christine Thielen (1999). *Guidelines für das Tagging deutscher Textcorpora mit STTS*. Tech. rep. Universität Stuttgart, Institut fur maschinelle Sprachverarbeitung; Universität Tübingen, Seminar für Sprachwissenschaft. URL: http://www.sfs.uni-tuebingen.de/resources/stts-1999.pdf (cit. on p. 28).

Schmid, Helmut (1995). "Improvements in part-of-speech tagging with an application to German". In: *Proceedings of the ACL SIGDAT-Workshop*. Dublin, Ireland. URL: http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/data/tree-tagger2.pdf (cit. on p. 9).

Schmidt, Thomas and Kai Wörner (2014). "EXMARaLDA". In: *Handbook on Corpus Phonology*. Ed. by Ulrike Gut Jacques Durand and Gjert Kristoffersen. Oxford University Press, pp. 402–419. URL: http://ukcatalogue.oup.com/product/9780199571932.do (cit. on p. 9).

Seufert, Stephan, Avishek Anand, Srikanta Bedathur, and Gerhard Weikum (2013). "Ferrari: Flexible and efficient reachability range assignment for graph indexing". In: *IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, pp. 1009–1020 (cit. on pp. 14, 119).

Sprenger, Stefan, Steffen Zeuch, and Ulf Leser (2017). "Cache-Sensitive Skip List: Efficient Range Queries on Modern CPUs". In: *Data Management on New Hardware*. Ed. by Spyros Blanas, Rajesh Bordawekar, Tirthankar Lahiri, Justin Levandoski, and Andrew Pavlo. Cham: Springer International Publishing, pp. 1–17. ISBN: 978-3-319-56111-0 (cit. on pp. 114, 123).

Stede, Manfred and Arne Neumann (May 2014). "Potsdam Commentary Corpus 2.0: Annotation for Discourse Research". In: *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14)*. Reykjavik, Iceland: European Language Resources Association (ELRA). ISBN: 978-2-9517408-8-4. URL: http://www.lrec-conf.org/proceedings/lrec2014/pdf/579_Paper.pdf (cit. on p. 99).

Stehouwer, Herman, Matej Durco, Eric Auer, and Daan Broeder (2012). "Federated search: Towards a common search infrastructure". In: *LREC 2012: 8th International Conference on Language Resources and Evaluation*. European Language Resources Association (ELRA), pp. 3255–3259. URL: http://hdl.handle.net/11858/00-001M-0000-000F-9E1D-8 (cit. on p. 124).

Steiner, Ilona and Laura Kallmeyer (2002). "VIQTORYA–A Visual Query Tool for Syntactically Annotated Corpora." In: *LREC 2002 Third International Conference on Language Resources and Evaluation*. Las Palmas, Spain. URL: http://lrec.elra.info/proceedings/lrec2002/pdf/116.pdf (cit. on p. 16).

Steland, Ansgar (Nov. 17, 2016). *Basiswissen Statistik*. 4. Auflage. Springer. ISBN: 978-3-662-49947-4. DOI: 10.1007/978-3-662-49948-1 (cit. on p. 109).

Telljohann, Heike, Erhard Hinrichs, Sandra Kübler, Heike Zinsmeister, and Kathrin Beck (Nov. 2009). *Stylebook for the Tübingen Treebank of Written German (TüBa-D/Z)*. Tech. rep. Universität Tübingen Seminar für Sprachwissenschaft. URL: http://www.sfs.uni-tuebingen.de/fileadmin/static/ascl/resources/tuebadz-stylebook-0911.pdf (cit. on pp. 15, 99).

Trissl, Silke (2012). "Cost-based optimization of graph queries in relational database management systems". PhD thesis. Humboldt-Universität zu Berlin, Mathematisch-Naturwissenschaftliche Fakultät II. DOI: 10.18452/16544 (cit. on p. 14).

Vanroy, Bram, Vincent Vandeghinste, and Liesbeth Augustinus (2017). "Querying large treebanks: Benchmarking GrETEL indexing". In: *Computational Linguistics in the Netherlands Journal* 7, pp. 145–166 (cit. on p. 122).

Voormann, Holger and Ulrike Gut (2008). "Agile corpus creation". In: *Corpus Linguistics and Linguistic Theory* 4.2, pp. 235–251 (cit. on p. 120).

Walter, Maik (May 2015). *Märchenkorpus Version 1.0*. Humboldt-Universität zu Berlin. http://www.textbewegung.de/. http://hdl.handle.net/11022/0000-0000-8211-9 (cit. on p. 99).

Wilkinson, Kevin, Craig Sayers, Harumi Kuno, and Dave Reynolds (2003). "Efficient RDF Storage and Retrieval in Jena2". In: *Proceedings of the First International Conference on Semantic Web and Databases*. SWDB'03. Berlin, Germany: CEUR-WS.org, pp. 120–139. URL: http://dl.acm.org/citation.cfm?id=2889905.2889914 (cit. on p. 14).

Wood, Peter T (2012). "Query languages for graph databases". In: *ACM SIGMOD Record* 41.1, pp. 50–60 (cit. on p. 11).

Wynne, Martin (2008). "Corpus Linguistics". In: ed. by Anke Lüdeling and Merja Kytö. Vol. 1. Handbooks of Linguistics and Communication Science (HSK) 29. Mouton de Gruyter. Chap. Searching and concordancing, pp. 707–737 (cit. on pp. 1, 14, 15).

Yildirim, Hilmi, Vineet Chaoji, and Mohammed J Zaki (2010). "Grail: Scalable reachability index for large graphs". In: *Proceedings of the VLDB Endowment* 3.1-2, pp. 276–284 (cit. on pp. 14, 119).

Zeldes, Amir (Apr. 2016a). *ANNIS User Guide - Version 3.4.3*. URL: http://corpus-tools.org/annis/resources/ANNIS_User_Guide_3.4.3.pdf (cit. on pp. 26, 29).

– (2016b). "The GUM corpus: creating multilayer resources in the classroom". In: *Language Resources and Evaluation*, pp. 1–32. ISSN: 1574-0218. DOI: 10.1007/s10579-016-9343-x (cit. on pp. 3, 88, 99).

Zinsmeister, Heike, Marc Reznicek, Julia Ricart Brede, Christina Rosén, and Dirk Skiba (Dec. 2012). "Das Wissenschaftliche Netzwerk „Kobalt-DaF"". In: *Zeitschrift für germanistische Linguistik* 40.3, pp. 457–458. DOI: 10.1515/zgl-2012-0030 (cit. on p. 99).

Zipser, Florian (2009). "Entwicklung eines Konverterframeworks für linguistisch annotierte Daten auf Basis eines gemeinsamen (Meta-)modells". MA thesis. Humboldt-Universität zu Berlin. URL: https://hal.archives-ouvertes.fr/hal-00606102 (cit. on pp. 21, 22).

– (2012). *Salt Model Guide*. SFB 632 Information Structure. URL: https://github.com/korpling/salt/raw/master/gh-site/doc/salt_modelGuide.pdf (cit. on pp. 21, 24).

– (Oct. 2014). "SaltNPepper und das Formatpluriversum". In: *LAUDATIO Workshop*. Humboldt-Universität zu Berlin. Berlin. DOI: 10.5281/zenodo.17557 (cit. on p. 9).

Zipser, Florian and Laurent Romary (2010). "A model oriented approach to the mapping of annotation formats using standards." In: *Workshop on Language Resource and Language Technology Standards, LREC 2010*. URL: http://hal.archives-ouvertes.fr/inria-00527799/ (cit. on pp. 10, 21).

Zipser, Florian, Amir Zeldes, Julia Ritz, Laurent Romary, and Ulf Leser (Feb. 2011). "Pepper: Handling a multiverse of formats". In: *33. Jahrestagung der Deutschen Gesellschaft für Sprachwissenschaft (DGfS 2011, Postersession der Sektion Computerlinguistik)*. Göttingen. DOI: 10.5281/zenodo.15638 (cit. on p. 9).

Acronyms

API Application Programming Interface 14, 27, 48, 84, 85, 120

AQL ANNIS Query Language 8, 12, 14, 17, 26–33, 35, 37, 38, 40–43, 45–50, 56, 60, 67, 70, 80, 82, 83, 88–91, 94, 97–99, 102, 106, 117, 118, 120–125, 127

CLARIN Common Language Resources and Technology Infrastructure 124

COTS commercial off-the-shelf 118–120

CQP Corpus Query Processor 14, 15, 17, 123

CTE Common Table Expression 119

CWB IMS Open Corpus Workbench 14, 15, 17, 18, 117, 120

DAG Directed Acyclic Graph 15, 32, 33, 62, 87, 88, 111, 143

DBMS database management system 32, 35

DFS Depth-First-Search 59, 61, 63, 87, 88, 145

DNF disjunctive normal form 48

DSL Domain Specific Language 7, 14, 27

GPL general-purpose programming language 27

ISO International Organization for Standardization 10, 124

JSON JavaScript Object Notation 48, 84, 98

KWIC Keyword in Context 3, 6

LAF Linguistic Annotation Framework 10

LHS left-hand side 29–32, 49–51, 67, 68, 72–77, 79–83, 89–91, 94–96, 102, 104, 107, 123

NLP Natural Language Processing 2

PTB Penn Treebank Bracket 9, 11, 15

RDBMS Relational database management system 13, 14, 46

RDF Resource Description Framework 11–14, 124

RHS right-hand side 29–32, 49–51, 67, 68, 72–77, 79–83, 89, 91, 94–96, 107

RST Rhetorical Structure Theory 3, 88

SIMD Single instruction, multiple data 93, 94, 96, 114, 123

SPARQL SPARQL Protocol and RDF Query Language 12

SQL Structured Query Language 11, 14, 16, 32, 33, 35, 36, 46, 111, 119

STL Standard Template Library 53, 60

STTS Stuttgart-Tübingen-Tagset 28

TPC Transaction Processing Performance Council 98

URI Uniform Resource Identifier 12

XML Extensible Markup Language 9, 27

List of Figures

List of Algorithms

List of Tables

Selbständigkeitserklärung

Hiermit erkläre ich, die Dissertation selbstständig und nur unter Verwendung der angegebenen Hilfen und Hilfsmittel angefertigt zu haben. Ich habe mich nicht anderwärts um einen Doktorgrad in dem Promotionsfach beworben und besitze keinen entsprechenden Doktorgrad. Die Promotionsordnung der Mathematisch-Naturwissenschaftlichen Fakultät, veröffentlicht im Amtlichen Mitteilungsblatt der Humboldt-Universität zu Berlin Nr. 126/2014 am 18.11.2014, habe ich zur Kenntnis genommen.

Berlin, den 19.02.2018 Thomas Krause